AD-A094 408 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/6 9/2
DESIGN AND ANALYSIS OF UPDATE MECHANISMS OF A DATABASE COMPUTER--ETC(U)
JUN 80 D K HSIAO, M J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-80-3 NL

1 OF 2
AD A
094408

TECHNICAL REPORT SERIES

A

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNI

OSU-CISRC-TR-80-3

DESIGN AND ANALYSIS OF UPDATE MECHANISMS OF A DATABASE COMPUTER (DBC)

by

David K. Hsiao & M. Jaishankar Menon

Work Performed Under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

June 1980

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OSU-CISRC-TR-80-3	2. GOVT ACCESSION NO. AD-A094408	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Design and Analysis of Update Mechanisms of a Database Computer (DBC)		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David K. Hsiao M. Jaishankar Menon		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Naval Research Information Systems Program Washington, D. C. 20360		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE June 9, 1980 13. NUMBER OF PAGES 132
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Scientific Officer	DCC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

APPROVED FOR PUBLIC RE
DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Database computer, update mechanisms, track-size buffers, insert-in-parallel, blocking, analytical model, queueing analysis.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report shows how the process of update is carried out in the database computer (DBC) which is a specialized back-end database machine capable of managing data of 10**10 bytes in size. Since DBC might often have to be used in an update-intensive environment (that is, an environment where many update, delete and insert commands and only a few retrieve commands are issued), we have indicated throughout this report, the kind of architectural enhancements which will provide good performance in an update-intensive environment.

Perhaps the most important enhancement that affects the performance of all

the four types of requests in DBC (retrieve, delete, insert and update) is the incorporation of a track-size buffer with each TIP. The advantages that accrue as a result of the incorporation are clearly demonstrated in the various sections of the report. For example, the process of compaction, which originally took 487 revolutions of the disk device now only takes one revolution of the disk device and one read-through of the buffer. Similarly, it is shown how an update request can be handled in two read-throughs of the sequential track-size buffer. This is a substantial improvement over the 16 revolutions that will be necessary to do an update without the use of track-size buffers. With respect to insertion requests, an important enhancement is the addition of an insert-in-parallel capacity. That is, records do not have to be inserted into MM of DBC one record at a time. Rather, all the TIPs can be inserting records at the same time.

We have also isolated and studied in this report the problem of clashing, i.e., requests being blocked by an update which has not been completely executed. The execution of the blocked requests must be stayed until the blocking update is executed completely.

Throughout the report, we have always substantiated our claims of performance improvement by using an analytical model to come up with quantitative figures of the data loop throughout. By using the model, we have also shown how a database administrator (DBA) can control the throughput achievable in DBC.

TABLE OF CONTENTS

Table of Contents Continued

ABSTRACT

This report shows how the process of update is carried out in the data-computer (DBC) which is a specialized back-end database machine capable of managing data of 10**10 bytes in size. Since DBC might often have to be used in an update-intensive environment (that is, an environment where many update, delete and insert commands and only a few retrieve commands are issued), we have indicated throughout this report, the kind of architectural enhancements which will provide good performance in an update-intensive environment.

Perhaps the most important enhancement that affects the performance of all the four types of requests in DBC (retrieve, delete, insert and update) is the incorporation of a track-size buffer with each TIP. The advantages that accrue as a result of the incorporation are clearly demonstrated in the various sections of the report. For example, the process of compaction, which originally took 487 revolutions of the disk device now only takes one revolution of the disk device and one read-through of the buffer. Similarly, it is shown how an update request can be handled in two read-throughs of the sequential track-size buffer. This is a substantial improvement over the 16 revolutions that will be necessary to do an update without the use of track-size buffers. With respect to insertion requests, an important enhancement is the addition of an insert-in-parallel capacity. That is, records do not have to be inserted into MM of DBC one record at a time. Rather, all the TIPs can be inserting records at the same time.

We have also isolated and studied in this report the problem of clashing, i.e., requests being blocked by an update which has not been completely executed. The execution of the blocked requests must be stayed until the blocking update is executed completely.

Throughout the report, we have always substantiated our claims of performance improvement by using an analytical model to come up with quantitative figures of the data loop throughput. By using the model, we have also shown how a database administrator (DBA) can control the throughput achievable in DBC.

1. BACKGROUND

The database computer (DBC) is a specialized back-end computer whish is capable of managing data of 10^{10} bytes in size and supporting known data models such as relational, network, hierarchical and attribute-based models. All operations performed by DBC are concerned with one or more of the following four aspects - searching and retrieval, security, clustering and updating. A number of papers [1-6] are available that motivate the design of DBC and discuss the search and retrieval aspects in some detail. A description of the security and clustering mechanisms of DBC and of the concepts that form the basis for these mechanisms appears in [7]. In this report, we intend to demonstrate how update is carried out in DBC and to make some suggestions on how the performance of DBC may be improved in an environment which is update-intensive. An update-intensive environment is one in which a large number of the requests issued to the database are insert, delete and update commands. That is, there are only a few **retrieve** commands issued to the database. Fundamental to our discussion is an understanding of the built-in data model and overall architecture of DBC and these are dealt with in the following sub-sections.

1.1 DBC Data Model

The smallest unit of data in DBC is a keyword which is an attribute-value pair, where the attribute may represent the type, quality, or characteristic of the value. Information is stored in and retrieved from DBC in terms of records; a record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(<Relation, EMP>,<Job, MGR>,<Dept, TOY>,<Salary,3ØØØØ>).

The record consists of four keywords. The value of the attribute Dept, for instance, is TOY.

DBC recognizes several kinds of keywords: simple, security and clustering. Simple keywords are intended for search and retrieval purposes. Security keywords are used for access control. Clustering keywords are utilized for placing records with a high probability of being retrieved together in close proximity. A discussion of security and clustering keywords appears in [7] and will not be reproduced here.

A keyword predicate, or simply predicate, is of the form (attribute, relational operator, value). A relational operator can be one of [=, ≠, >, ≥, <, ≤]. A keyword K is said to satisfy a predicate T if the attribute of K is identical to the attribute in T and the relation specified by the relational operator of T holds between the value of K and the value in T. For example, the keyword <Salary,15ØØØ> satisfies the predicate (Salary > 1ØØØØ).

A query conjunction, or simply conjunction, is a conjunction of predicates. An example of a query conjunction is:

$$(Salary>25000) \wedge (Dept=TOY) \wedge (Name = JAI).$$

We say that a record satisfies a query conjunction if the record contains keywords that satisfy every predicate in the conjunction.

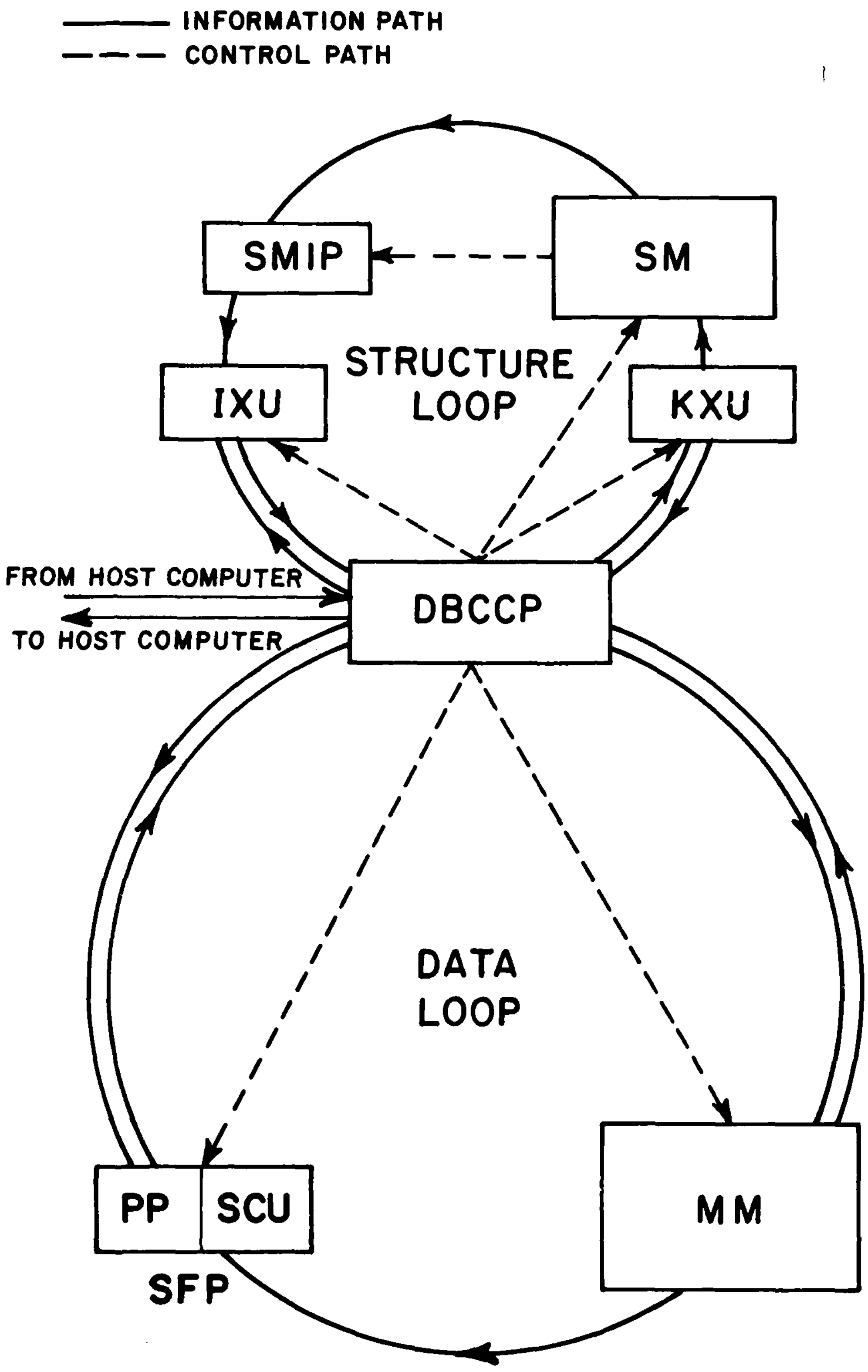
A query is a Boolean expression of predicates in the disjunctive normal form. Thus, a query is a disjunction of query conjunctions. An example of the types of queries that may be recognized by DBC is as follows:

$$((Dept=TOY) \wedge (Salary<1ØØØØ)) \vee ((Dept=BOOK) \wedge (Salary>5ØØØØ)).$$

If the above query (consisting of two conjunctions) refers to a file of employees of a department store, then it will be satisfied by records of employees working either in the toy department and making less than $1Ø,ØØØ, or working in the book department and making more than $5Ø,ØØØ. We say that a record satisfies a query, if the record satisfies at least one of the conjunctions in the query. Thus, we can refer to a set of records that satisfy a query. A query, as defined above, is used not only to retrieve, delete and update the set of records that satisfy the query, but also to specify protection requriements [7].

1.2 DBC Architecture

Figure 1 shows the schematic architecture of DBC. It consists of two loops of memories and processors, namely the structure loop and the data loop. The data loop is composed of two components: the mass memory (MM), and the security filter processor (SFP). MM is the repository of the database and is made of modified moving-head disks where all the tracks of a cylinder may be read in parallel in a single disk revolution. This modification is termed tracks-in-parallel-readout. In addition, the mass memory of DBC is content-addressable. Given a cylinder number and a query conjunction, it is possible to content-search the entire cylinder 'on the fly' for records that satisfy the query conjunction. The MM is described in detail in Section 2 of this report.



DBCCP:	Data Base Command Control Processor
KXU:	Keyword Transform Unit
SM:	Structure Memory
SMIP:	Structure Memory Information Processor
IXU:	Index Translation Unit
MM:	Mass Memory
SFP:	Security Filter Processor
SCU:	Security and Clustering Unit
PP:	Post Processor

Figure 1. The Architecture of DBC

The structure loop is composed of four components: the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). KXU converts keywords into their internal representations. SM is primarily used to store, retrieve and update the indices of the database. Indices are maintained in SM as a directory. Each entry in the directory consists of a keyword or a keyword descriptor followed by a set of indices. A keyword descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. An example of a keyword descriptor is:

$$((\text{Salary}) \geq 2{,}000) \wedge (\text{Salary} \leq 10{,}000)).$$

More simply, this is written as follows:

$$(2{,}000 \leq \text{Salary} \leq 10{,}000).$$

Thus, a keyword descriptor is an attribute (Salary) and a range of values ($2,000 - $10,000) for that attribute. A keyword K satisfies a keyword descriptor KD, if the attribute of vK is identical to the attribute KD and the value of K lies within the range of values or KD. An index is a pair of the form (cylinder number, security atom number) [7]. The cylinder number indicates where in mass memory records with keywords satisfying the keyword descriptor may be found and the security atom number indicates the access privileges accorded to these records. Any keyword that appears as part of a directory entry or satisfies a keyword descriptor in SM is called a directory keyword. For example, security keywords and clustering keywords are automatically defined to be directory keywords and they always satisfy certain keyword descriptors in SM. However, not all simple keywords are directory keywords. Non-directory keywords are mainly used by MM and SFP for record comparison and sorting purposes. SMIP is responsible for performing set intersections on the indices retrieved by SM. IXU is used to decode the indices output by SMIP. These four components are designed to operate concurrently in a pipeline fashion. The hardware organization, details and design philosophy of these components are documented in [5].

The database command and control processor (DBCCP) regulates the operations of both the structure and data loops and interfaces with the front-end host computer. It processes all DBC commands received from the front-end host computer, schedules their execution on the basis of the command type and priority, and routes the response to the front-end host computer. Additionally, it makes use of SFP to search the tables needed for the enforcement of security and clustering of records [7].

1.3 Control Flow During Command Execution

Figure 2 shows how certain commands are executed in DBC. Basically, these commands forwarded from the front-end host computer (in pre-determined formats), are recognized by DBCCP as either access commands or preparatory commands. Access commands are those that require DBC to access the mass memory; preparatory commands precede and follow access commands and convey important house-keeping information. Access commands go through a security check. Having undergone security checks, access commands are translated by DBCCP into orders that can be processed by the mass memory. During translation, an access command involving insertion activates the clustering mechanism in DBCCP. This clustering mechanism determines the mass memory cylinder into which the record must be inserted. Records retrieved by the mass memory (as a result of the execution of orders and the supply of cylinder numbers provided by DBCCP) are transmitted to the sorter. The sorter allows groups of retrieved records to be sorted on the basis of some attribute, or joined with other groups of records (a join being a relational equality join) [8,9].

1.4 Organization of This Report

In Section 2, we shall give an overview of the architecture of MM. Sections 3, 4, and 5 are devoted to describing how the processes of insertion, deletion and update are accomplished in DBC. More specifically, we shall describe the process of insertion of records in Section 3, while Section 4 will be devoted to a discussion of the process of deletion of records. In Section 5, we discuss the method used in DBC to update records. In each of the last three sections, we shall use an analytical model of the data loop of DBC in order to evaluate the actual performance improvements achieved as a result of certain suggested changes. Finally, in Section 6, we present an integrated picture of DBC with all the changes that have been suggested in previous sections.

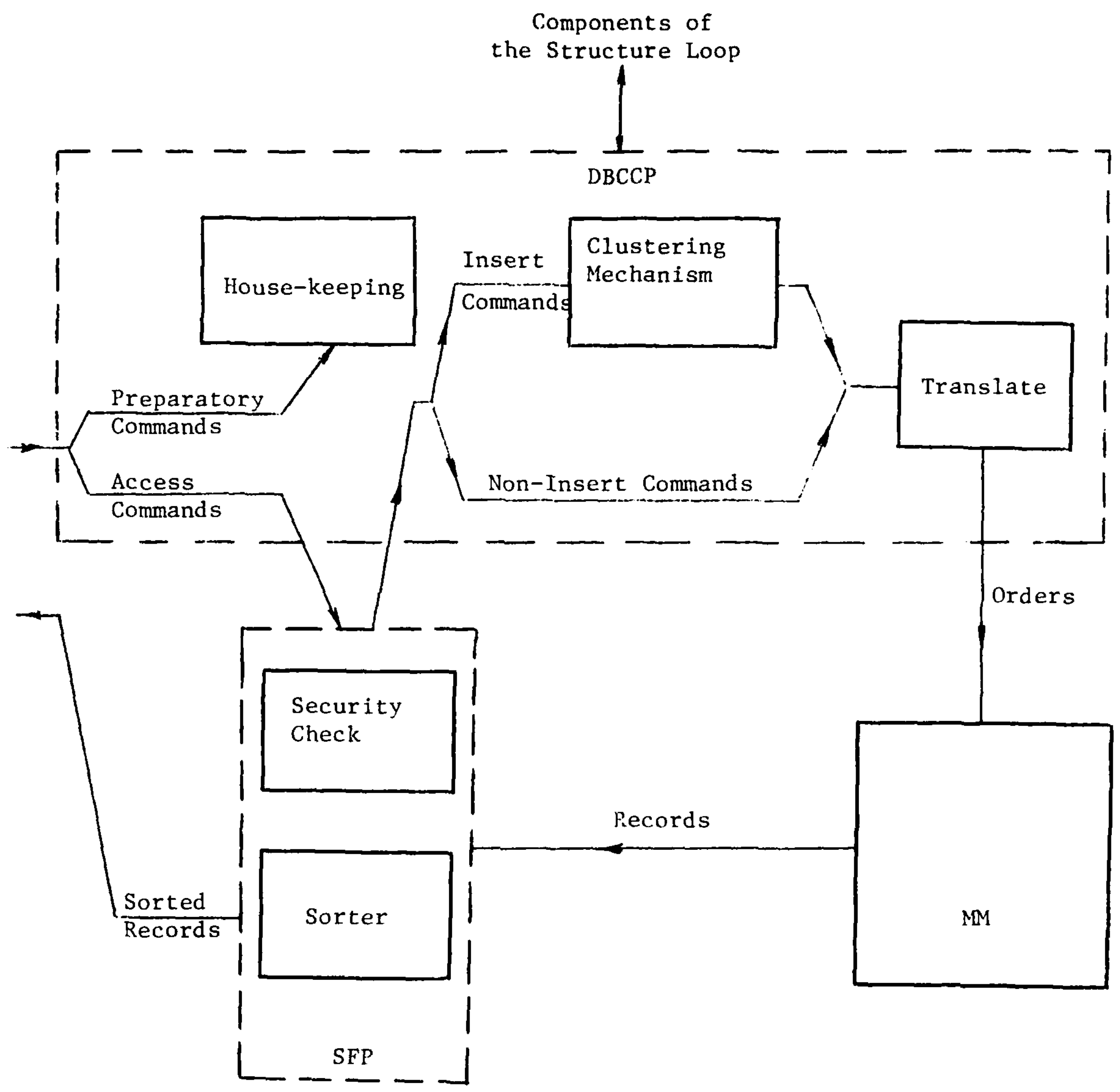


FIGURE 2. Access Command Execution in the Data Loop

2. THE MASS MEMORY

Since it is the intention of this report to study the updating of records stored in the mass memory (MM), an understanding of the organization of MM is essential. MM may conveniently be thought of as consisting of two parts. The first part is the repository of the database. The second part is the set of processors that are used to search, retrieve and update records stored in the repository. We begin by describing the nature of the repository of the database and also the reasons behind our choice of such a repository. Later, we go on to talk about the architectual features of the processing elements that are used to manipulate the database. The philosophy upon which the design decisions regarding these processing elements is predicated is not discussed in this section. That will be done in later sections.

2.1 The MM Design Philosophy

For our design of the first part of the MM, we have chosen the moving-head disk as the storage medium. Our discussion in favor of this technology is based on the following reasons: First, moving-head disk technology is well entrenched and is unlikely to be replaced in this century [10]. Second, the cost per bit of this storage medium is about 5 millicents, providing an order of magnitude reduction in the storage costs over fixed-head disks or their electronic replacements. Third, it is possible to enhance the processing rate of a conventional moving-head disk by activating all the read/write heads available in the access mechanism. The achievement of this parallel readout does not involve any technological breackthrough. Ampex Corp. has modified one of their 9300 series 300 megabyte disks to offer the transfer of up to 9 disk tracks in parallel [14]. In such a modification, the information on the tracks of a cylinder can be searched 'on the fly' by employing a set of processing elements, one for each track. Each of the processing elements would be responsible for searching the information read from the corresponding track.

The above scheme provides content-addressability on a cylinder basis. We can thus think of the moving-head disk as being partitioned into cylinders, with content-addressability provided within each cylinder In our subsequent discussions, we shall refer to a partition of the MM as a cylinder or a minimal access unit (MAU).

2.2 Toward an Intelligent Mass Memory

We note that data manipulation requests sent to MM can be highly content-oriented. Also, data manipulation requests identify the data to be manipulated by means of queries. Since a query is a disjunction of conjunctions, MM must ensure that a record in an MAU (whose MAU number is supplied by the structure loop) sati

it as a valid data item. This can be done easily if both the query and the record are handled in their natural formats.

Since a record in a track is to be compared to a query conjunction 'on the fly', an arbitrary arrangement of keywords within the record and an arbitrary arrangement of keyword predicates within the query conjunction can lead to processing delays of up to several revolutions of the disk. For example, consider the following record and query conjunction arrangement:

RECORD:(<1,x>,<7,y>,<4,z>)

QUERY CONJUNCTION: $(4 \geq z) \wedge (7 = y) \wedge (1 \leq x)$

Here, 1, 4, and 7 are attribute identifiers and x, y, and z are actual values. In the above situation, the first attribute identifier read by the processing element is 1. Since the first attribute identifier in the query conjunction is 4, the processing element has to wait until the last attribute-value pair is read before making a comparison. After the comparison has been successfully completed, the processing element must compare the value of the second attribute 7 in the query conjunction with the corresponding value in the record. However, since the disk device is a uni-directional device, the processing element must wait for one revolution before attempting to make the comparison. It is easy to see that it requires three revolutions of the disk device to process the above record against the given query conjunction. Thus, back-tracking on a disk device can be expensive in terms of processing time.

Alternatively, if the query conjunction is stored in a random access memory accessed by the processing element, then for each attribute identifier read by the read head, a full search of the query conjunction memory can be made to determine if the corresponding identifier is present in the conjunction. The main drawback of this scheme is that, as the query conjunction gets longer and longer, it becomes more and more difficult to undertake a full search of the query memory in the time taken to read an attribute identifier (typically 1.5 microseconds).

A solution to this problem lies in a carefully planned layout of the record and the query conjunction. The attribute-value pairs in a record are arranged in ascending order of the attribute identifiers. The predicates in the query conjunction are similarly arranged in ascending order of their attribute identifiers. The query conjunction is stored in a sequentially accessed memory called query memory. The processor reads a stream of keywords belonging to a query conjunction from the query memory. The two streams are compared in a bit-serial fashion. Whenever there is a match between the attribute identifier in the conjunction and the attribute identifier in the record, the values are compared to determine if the predicate is satisfied. If the attribute identifier in the record is less than the attribute identifier in the conjunction, then the processing element skips over the corresponding value to the next attribute identifier. If the attribute identifier in the record is greater than the attribute identifier in the conjunction, it is concluded that the record does not satisfy the conjunction. The above logic is repeated until all the predicates in the conjunction are satisfied or the processing element concludes that the record does not satisfy the conjunction.

2.3 The Organization of the MM

The overall organization of MM is shown in Figure 3. The database resides in volumes mounted on moving-head disk drives. It is desirable to have a one-to-one corresondence between the volumes and the drives; but this is not essential if the volumes are transferable. However, with disk technologies moving towards higher bit densities, mechanical tolerances will not allow frequent interchange of volumes between disk drives [10]. DBC design is independent of the above consideration. A volume is composed of 200-400 cylinders. Each cylinder consists of a set of tracks (usually in the range of 20-40); there is one track of a cylinder per disk surface. The access mechanism consists of a movable set of read/write heads, one per disk surface. The heads are moved in unison to access all the tracks of a cylinder. Data transfer to/from a cylinder is achieved by activating all the read/write heads concurrently. Although previous desings [11,12] have taken advantage of the fact that the read and write heads on a track could be positioned a short distance from each other, we do not favor such an arrangement. This is because, at high track densities (1000 tracks per inch or higher), the required mechanical tolerances for supporting separate read and write heads may well deprive the disk technology of much of the cost effectiveness brought about by the higher densities [13]. In our design, therefore, we assume the conventional

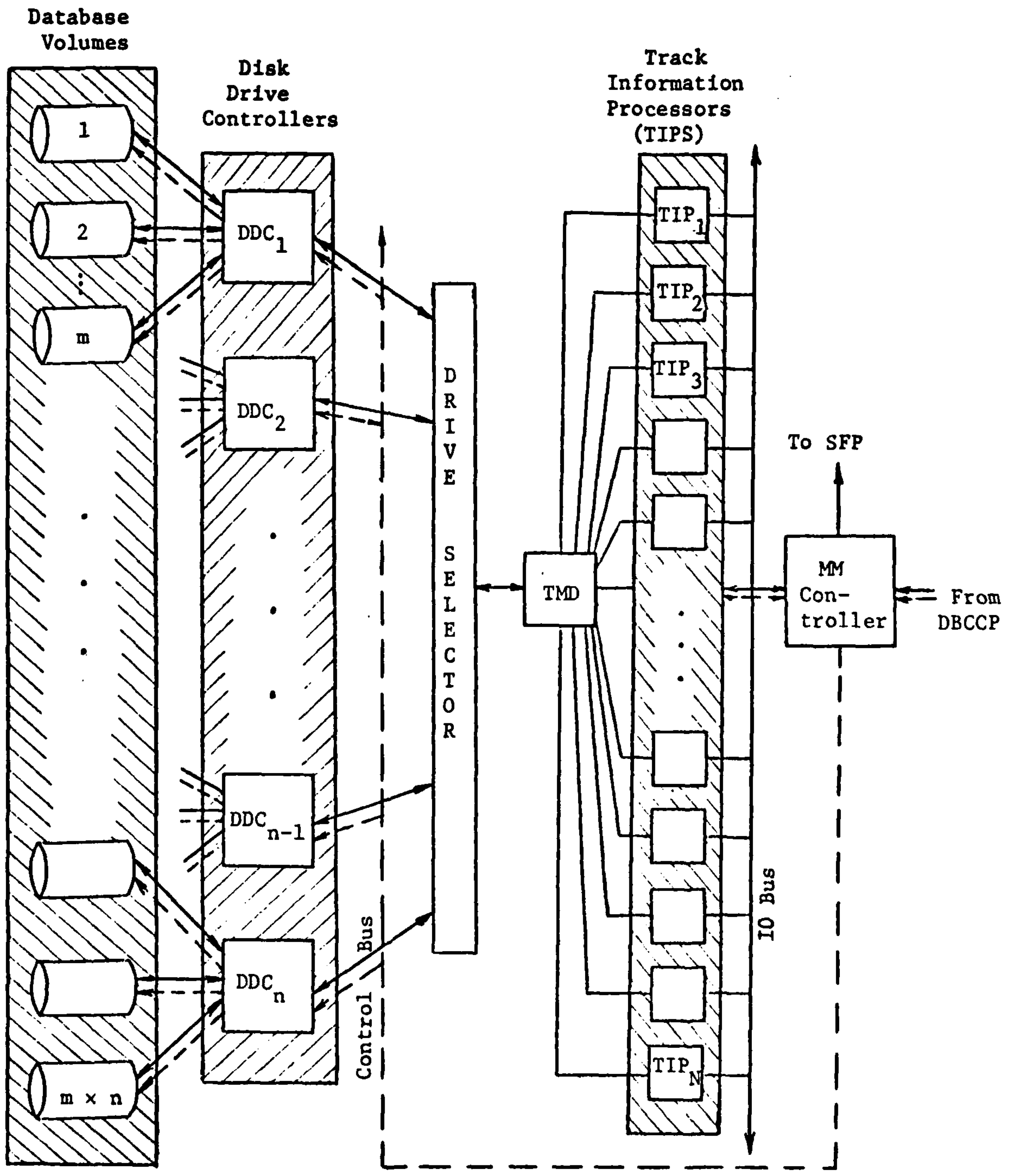


Figure 3. Organization of the Mass Memory (MM)

read/write mechanism. The implication of such a design is that MM can either be read from or written into at a given time. Reading and writing cannot be performed simultaneously.

Each MAU in the system is uniquely identified by a number known as its MAU address. A disk volume contains a set of consecutively addressed MAUs. The set of disk drives is partitioned into 8-16 drives for access and control purposes. Each such group is controlled by a _disk drive controller_ (DDC). The DDCs are controlled by the _mass memory controller_ (MMC). Data that are retrieved from the disk volumes are routed to a set of _track information processors_ (TIPs) by a drive selector and by a _track multiplexor/demultiplexor_ (TMD). The drive selector is controlled by the MMC. The TIPs can request the service of a bus called the IOBUS for transferring database information to the MMC. The IOBUS is also used by the MMC to send control information and data to the TIPs.

MM operates in two basic modes - the _normal mode_ and the _compaction mode_. In the normal mode, orders sent by DBCCP are decoded by the MMC and are queued according to the MAUs referenced by the orders. For each MAU for which a queue of orders exist, MMC requests the appropriate DDC (if free) to position the read/write heads to the cylinder corresponding to the MAU. When the MAU is found, the MMC sends the orders one at a time to the TIPs. While the TIPs are busy executing the orders, MMC can request the DDCs to position the read/write mechanisms to the MAU(s) residing on other volumes for which there are non-empty queues. Thus, the access time with respect to the MAU of one volume is at least partly overlapped by useful work performed by the TIPs on the MAU of another volume. The information retrieved by the TIPs from the database is sent to the SFP for further processing.

Records which are identified by a delete order under the normal mode are tagged by the TIPs for later removal during the compaction mode. When DBCCP orders MM to reclaim the space occupied by the records with deletion tags, MM enters the compaction mode. During the compaction mode, MAUs in which tagged records exist are accessed, and data in each of the tracks is read into MMC by the TIPs. MMC then writes back those records which are not tagged.

2.3.1 The Mass Memory Controller (MMC)

The mass memory controller is organized into two subcomponents: the _interface processor_ (IP) and the _mass memory monitor_ (MMM). The IP is responsible for interfacing with DBCCP, for maintaining the _database object_ descriptor table (DODT), and for maintaining MM orders in the mass memory order queues (MMOQ), and switching from normal to compaction mode. The mass memory monitor is responsible for scheduling orders to be executed with the help of the order queues (MMOQ), for issuing orders to the DDCs to position read/write heads, for initiating TIPs to execute the orders on the contents of a MAU and for keeping track of space availability in the MAUs.

2.3.1.1 Interface Processor (IP)

A. The Database Object Descriptor Table (DODT)

This table contains the database objects which are used as arguments of orders issued by DBCCP. Each object is identified by a unique identification tag assigned to it by DBCCP. A database object in this table could either be a query conjunction, a record or a pointer. The keywords in a query or a record are assumed to be sorted in ascending order of their attribute identifiers. This sorting is done before the query or record is sent to DBC. Since database objects are placed in the table only to be accessed later when the MM order is scheduled to be executed, there must be a rapid mechanism to locate and retrieve database objects from the table. The table is, therefore, organized in two parts - an associative memory (AM) and a random access memory (RAM). An entry in the AM has two fields - an object identification tag and a pointer to a location in the RAM. The RAM holds the database objects pointed to by the AM. The AM can be searched on the basis of database object identifiers; the response is the pointer to the RAM location where the corresponding database object is stored. In Figure 4, the organization of the table is shown.

B. Order Queues (OQ)

Order queues, as the name implies, are used to keep track of MM orders (sent by DBCCP) which are awaiting execution. There is one queue for every MAU for which one or more orders are awaiting execution. We shall often refer to all the orders awaiting execution on a particular MAU as a set of orders on that MAU. Two data structures are proposed in Figure 5 to manage order queues. The queue headers table (QHT) is used to carry information about the queues. More specifically, each entry in the QHT has three fields: the first field has status information about the availability of a MAU for processing. The second field contains the MAU address and the third field points to the first order to be executed on the MAU. The second data structure is called the order table (OT), which contains the orders themselves. The format of an order when it is received by MM is shown in Figure 6 and its format when it is stored in the order table is shown in Figure 7.

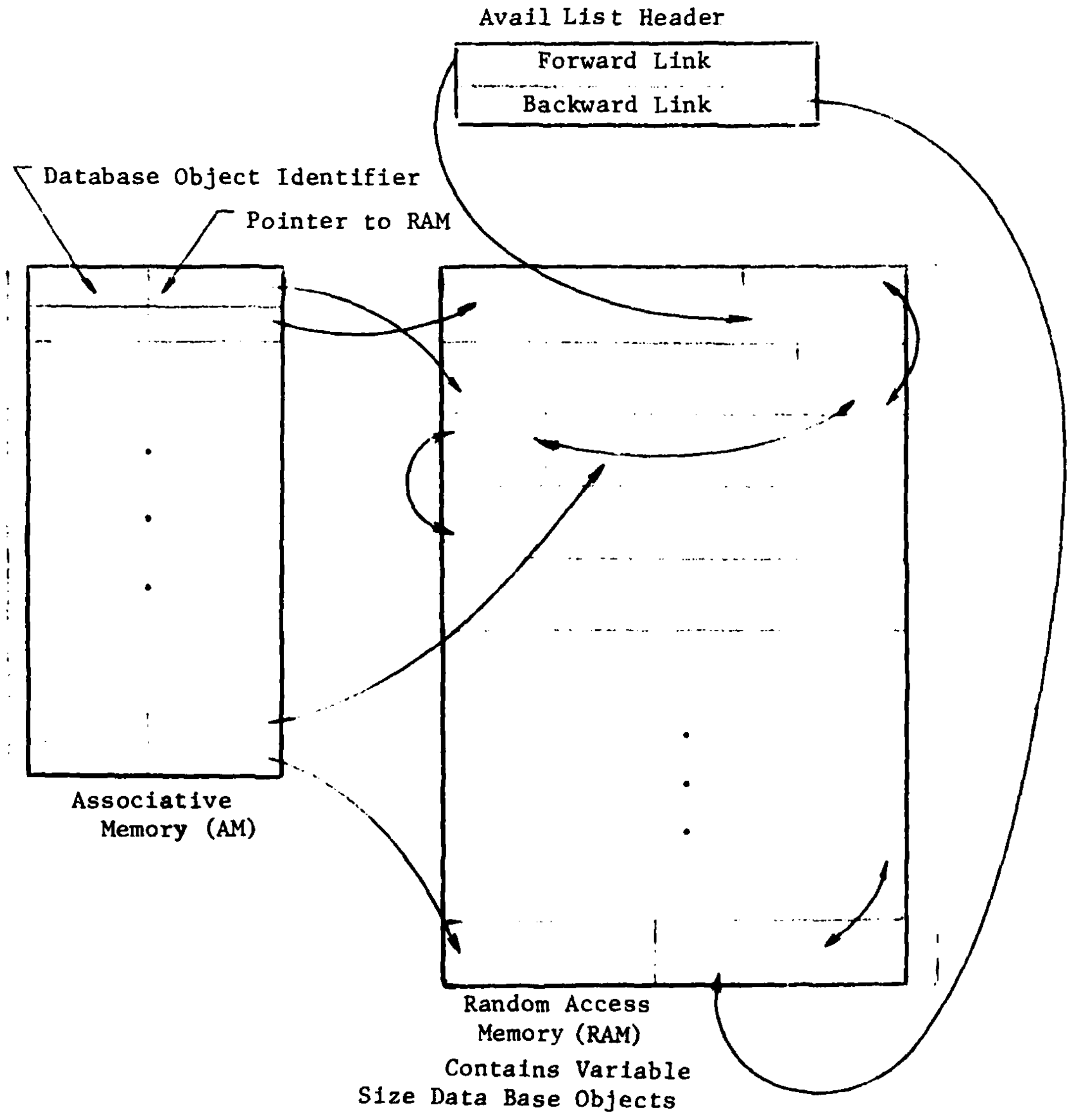
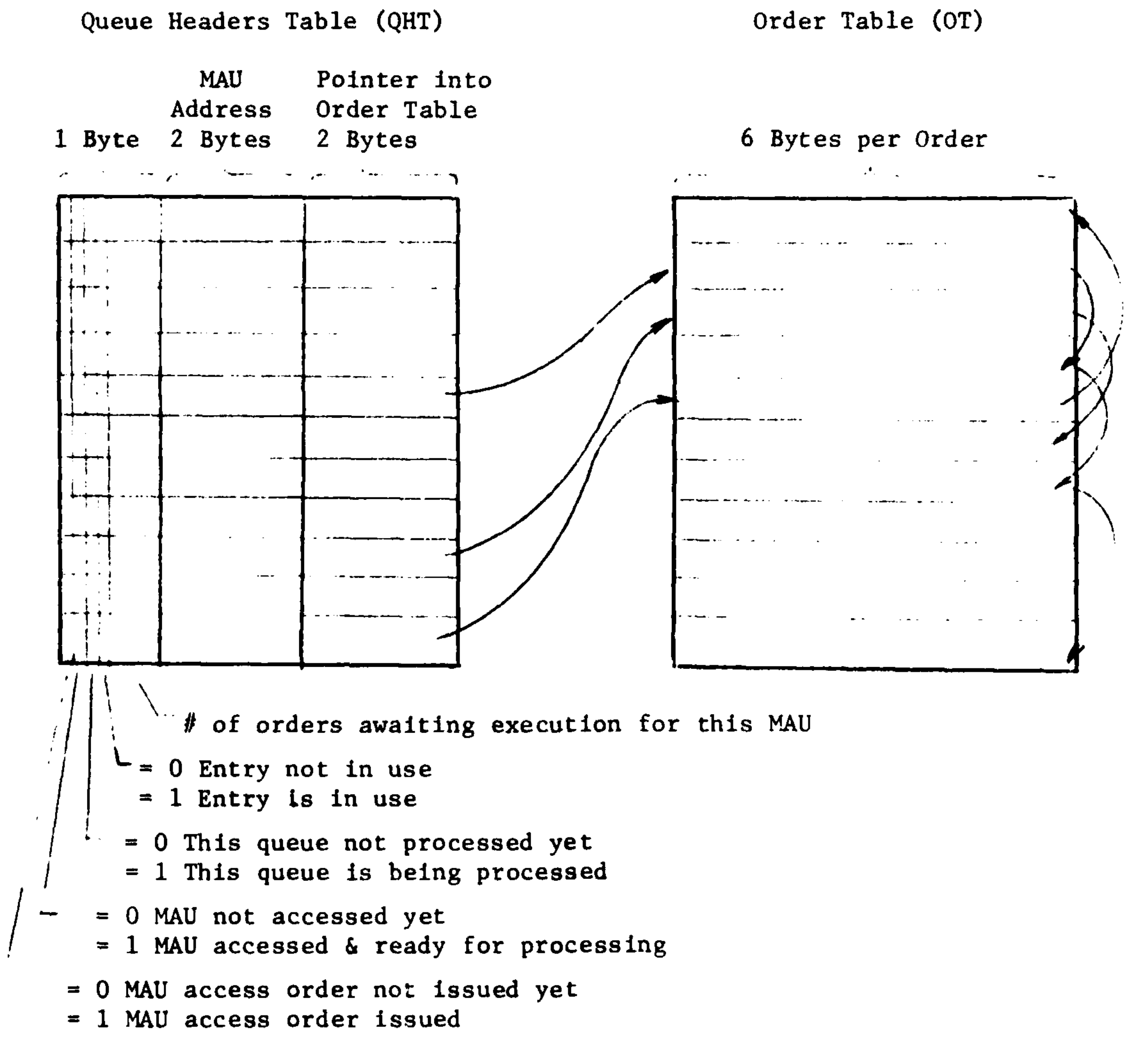

Avail List Header
Forward Link
Backward Link
Database Object Identifier
Pointer to RAM
Associative Memory (AM)
Random Access Memory (RAM)
Contains Variable Size Data Base Objects

Figure 4. Organization of the Database Object Descriptor Table (DODT)



Queue Headers Table (QHT)
Order Table (OT)
MAU Address 2 Bytes
Pointer into Order Table 2 Bytes
1 Byte
6 Bytes per Order
# of orders awaiting execution for this MAU
= 0 Entry not in use
= 1 Entry is in use
= 0 This queue not processed yet
= 1 This queue is being processed
= 0 MAU not accessed yet
= 1 MAU accessed & ready for processing
= 0 MAU access order not issued yet
= 1 MAU access order issued

Figure 5. Order Queues

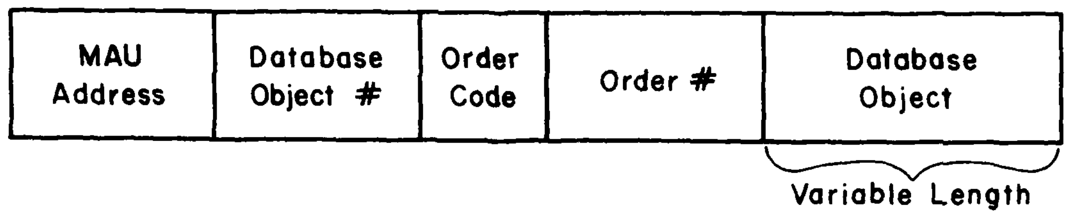


Figure 6. Format of a MM Order Sent by DBCCP

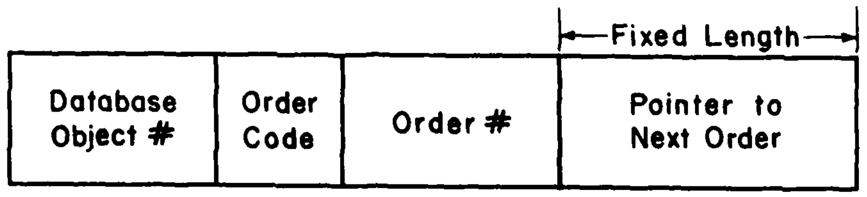


Figure 7. Format of a MM Order Stored in the Order Table (OT)

C. The IP Logic

IP executes Algorithm 1 (see Appendix 1) on receipt of an order from DBCCP. The algorithm explains, in detail, how the IP places orders into the various order queues.

2.3.1.2 The Mass Memory Monitor (MMM)

A. Mass Memory Deletion Table (MMDT)

The MMM maintains a deletion table to keep track of the MAUs in which there are records tagged for deletion. This table is created during the normal mode of operation and is used during the compaction mode to access the MAUs in which compaction must be performed. There is one entry in the MMDT for each such MAU. The first entry in the MMDT records the number of entries n that are currently in use. This is followed by the addresses of n MAUs.

B. The MMM Logic

The mass memory monitor controls the DDCs via the control bus (CBUS) (see Figure 3). The CBUS has an appropriate number of addres s lines by which each of the DDCs can be addressed to the exclusion of the others. The CBUS also carries status and control lines by which the MMM can control the DDCs and communicate with them. The MMM also controls the track information processors via the IOBUS. The IOBUS is operated in a master-slave mode with MMM assuming the master role and the TIPs assuming the slave roles. The IOBUS consists of bi-directional data lines over which data transfers between the TIPs and the MMM can take place, and status, address and control lines which enable the MMM to interrogate and activate the TIPs.

MMM executes two algorithms in the course of carrying out its functions outlined earlier. In these algorithms, all dialogues with the DDCs are carried over the CBUS and all dialogues with the TIPs are carried out over the IOBUS (see Appendix 1). Algorithm A continuously monitors the OHT with a view to keeping the TIPs and the disk drives busy. Algorithm B is responsible for the detailed dialogues with the TIPs after Algorithm A has found a MAU that has been accessed and is ready to be processed. Among other things, Algorithm B answers interrupts from the TIPs when they have output to be sent out of MM or when they have finished execution of an order. Once activated by Algorithm A, Algorithm B executes concurrently with Algorithm A, until the pending orders for the MAU have been executed by the TIPs.

2.3.1.3 The Hardware Organization of the MMC

The organization of the MMC is shown in Figure 8. The internal data bus (IDB) is the main data path inside the MMC. It connects all the table memories with the mass memory order argument buffer (MMOAB) and the mass memory data buffer (MMDB). The argument buffer is used to receive argument data of a MM order from the communication bus before they are transferred into the DODT. The data buffer is used primarily as a buffer between the IOBUS on which the TIPs place the retrieved data and the post processor bus PPBUS which transmits data to the SFP. The data buffer is also used during compaction as a stager between the IOBUS and the internal data bus. The interface processor (IP) logic is microcoded in ROM-1 and is executed by the microsequencer MC-1. It responds to request signals from the DBCCP and controls the transfer of data from and to the argument buffer. The MMM is implemented with two microsequencers and two control ROMs. Microsequencer MC-2 is responsible for executing Algorithm B of Appendix 1. It is responsible for controlling the activities of the TIPs, controlling the data transfers on the IOBUS, and data transfers to and from the MMDB. MC-2 also receives interrupt signals from the TIPs. The microsequencer MC-3 is responsible primarily for scanning the order queues, initiating MC-2 and controlling the DDCs. Finally, the bus arbiter is responsible for processing requests for control of and access to the IDB and resolving contentions for the control of the IDB.

2.4 The Flow of Information in the MM

In this section, we describe how information is routed from/to the disk volumes to/from the TIPs which process the information contained in the disk volumes.

As mentioned earlier, there is a single set of TIPs in MM. The number of TIPs in the set is equal to the number of tracks in a cylinder of a disk volume. At any given instant of time, the TIPs could be processing information from exactly one cylinder.

A set of disk drives is controlled by a *disk drive controller* (DDC). The DDC can initiate data transfer operations on any one of the drives controlled by it. DDC provides for a set of assembly/disassembly registers. There is also a set of input/output registers which are multiplexed by DDC before sending their contents to the drive selector. There is one pair of assembly/disassembly and input/output registers for each track of a cylinder (see Figure 9). The width of these registers is known as the data unit. These registers

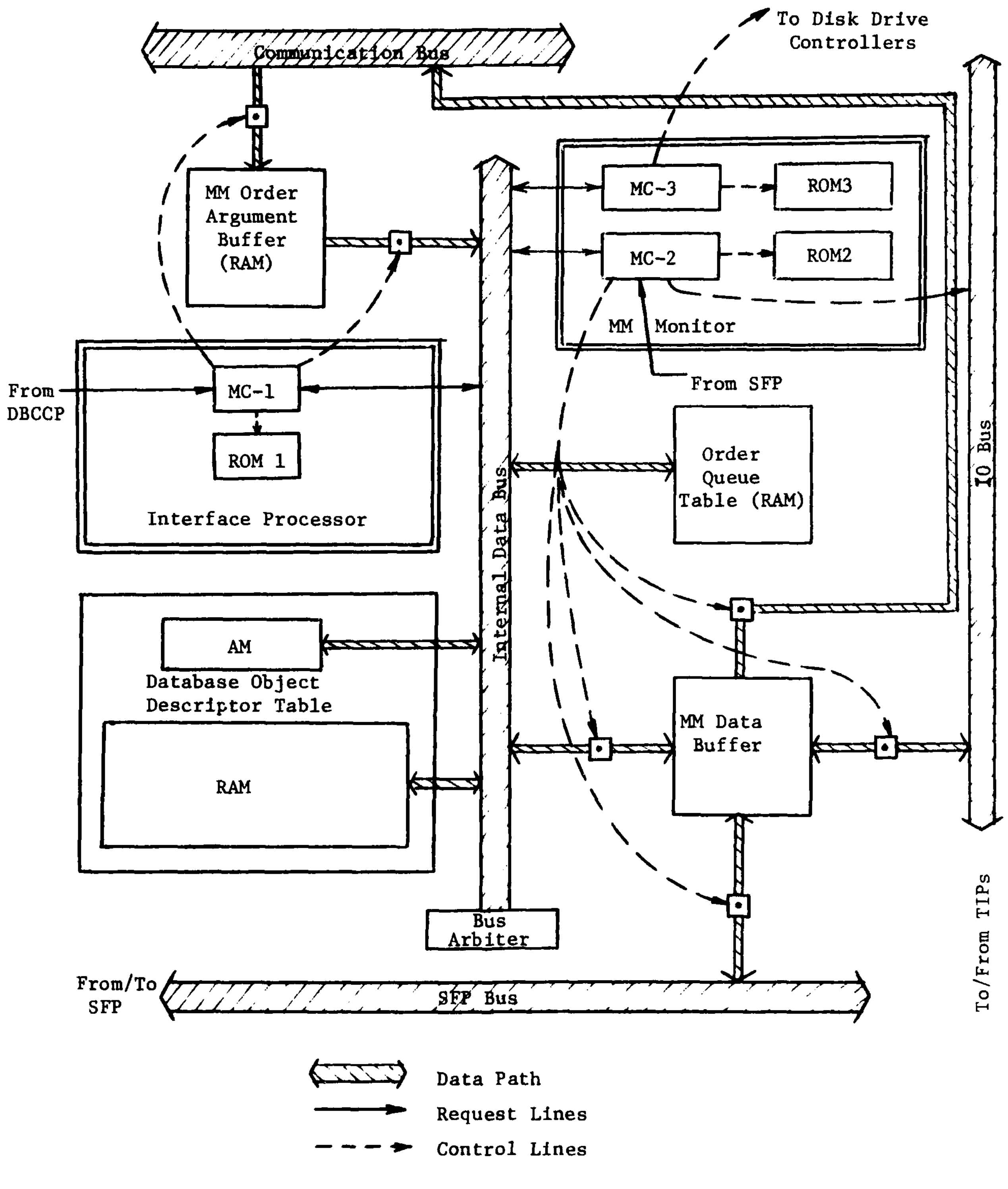


Figure 8. Organization of the Mass Memory Controller

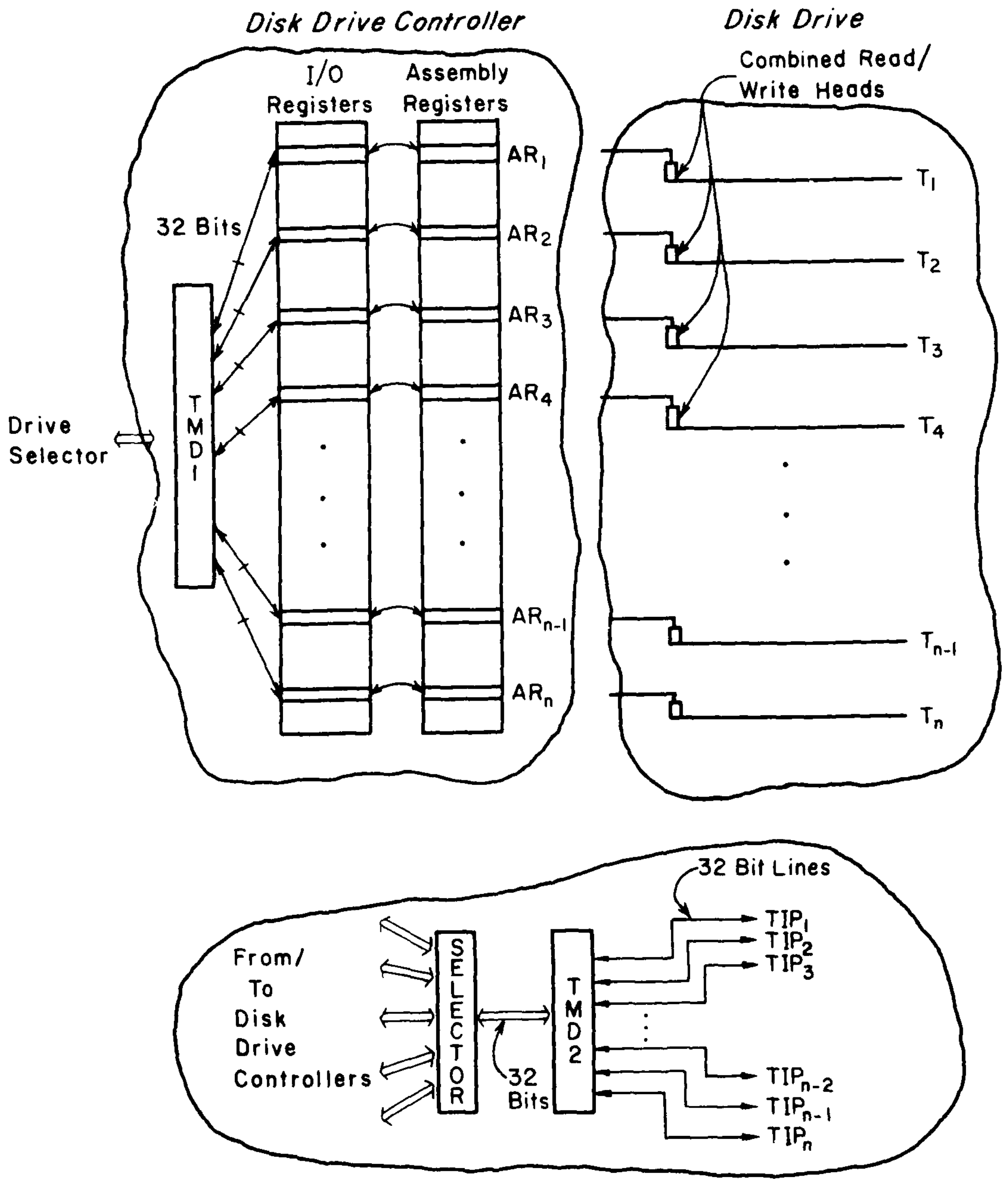


Figure 9. A Scheme to Route Information between Track Information Processors and the Tracks

serve the following purpose: during a read operation, data bits from the tracks are assembled into data units in the assembly registers: at the same time, the previous data units are held in the input/output registers are multiplexed (by the track multiplexor/demultiplexor TMD2) into a serial stream of data units and sent to the drive selector. During a write operation, the reverse operations take place. That is, the sequential stream of bits received from the drive selector are demultiplexed (by TMD1) and placed in the input/output registers of the DDC. At the same time, previous data units held in the assembly/disassembly registers are written onto the corresponding tracks.

During a read operation data units received by the drive selector from the appropriate DDC are allowed to pass through to the track multiplexor/demultiplexor (TMD2). The TMD (TMD2) then directs the data units to the appropriate TIP. During a write operation, the data units from the TIPs are collected by TMD2 and sent to the drive selector in a sequential stream. The drive selector then routes the data units to the DDC selected by MMC. Thus, we note, that not only is it possible to read out of all the tracks of a cylinder in parallel, it is also possible to write into them in parallel. We shall call this capability the <u>parallel-write-in</u> capability.

2.5 <u>The Track Information Processors</u>

A track information processor (TIP) is responsible for manipulating the contents of a track belonging to a MAU. The number of TIPs is equal to the number of tracks in a MAU and is usually in the range 20-40. The TIPs are capable of searching the tracks (of an MAU) for records satisfying a user query conjunction in one revolution of the rotating device.

2.5.1 <u>The Three Components of a TIP</u>

Each TIP has three sub-components - the <u>disk drive interface processor</u> (DIP), the <u>controller interface processor</u> (CIP), and the buffers for the query, retrieved information (records), track header information and communications. The DIP is responsible for receiving/transmitting data as demanded by TMD2 and carrying out the orders sent to it. The CIP is responsible primarily for communicating with the mass memory controller over the IOBUS. Such communication involves acceptance of orders and database objects from the mass memory controller and transfer of data retrieved by the DIP to the post processor via the IOBUS.

The communication buffer and the buffer for the track header information are small random access memories. The query memory is a sequential access memories. The query memory is a sequential access memory with a capacity to store the largest single query conjunction that may be encountered by MM (about 1 Kbytes). The record buffer is also a sequential access memory. This memory is divided in to a number of individually accessible segments. Each segment may be read out of or written into in a sequential manner. The motivation for dividing the record buffer into segments is as follows: While the DIP is extracting information from the track and placing it in one of the segments, the CIP can be transmitting previously extracted information present in one of the other segments to MMC over the IOBUS.

The readout rate of the query memory and the transfer rate of the record buffer should be high enough to keep up with the data transfer rate of the disk device. The organization of a TIP is shown in Figure 10. The format of the communication area between the CIP and the DIP is shown in Figure 11. The format of a track as perceived by a TIP is shown in Figure 12. Each of the TIPs utilizes a bit map to remember the positions of the records which were found to satisfy a search criterion during the execution of a delete order. Each record on the track is represented by a unique bit in the bit map. When a record is to be deleted, the corresponding bit is turned on. This bit map is stored at the beginning of a track. Before processing of a cylinder is to begin, the bit map in each of the tracks is read by the corresponding TIP. In processing a retrieve or update order, this bit map is consulted to ensure that no tagged records are retrieved. After the last order for an MAU has been executed (i.e., after the execution of a set of orders), the bit map is written back on the track. During the compaction mode of operation, this bit map is used to distinguish between tagged and untagged records. Each track is divided into a fixed number of sectors for the purpose of allocation. The first two sectors are used by the TIPs to store the bit map and other housekeeping information.

The disk drive interface processor (DIP) is a bit-slice processor capable of carrying out fast comparisons of attribute identifiers occurring in records stored on a track with those occurring in the query conjunction in a user request. It is also capable of comparing keyword values in the records with values associated with keyword predicates in the query conjunction of a user request. This enables the drive interface processor to carry out range searches. The control unit of the DIP is microprogrammed to interpret the orders sent to it by the MMC.

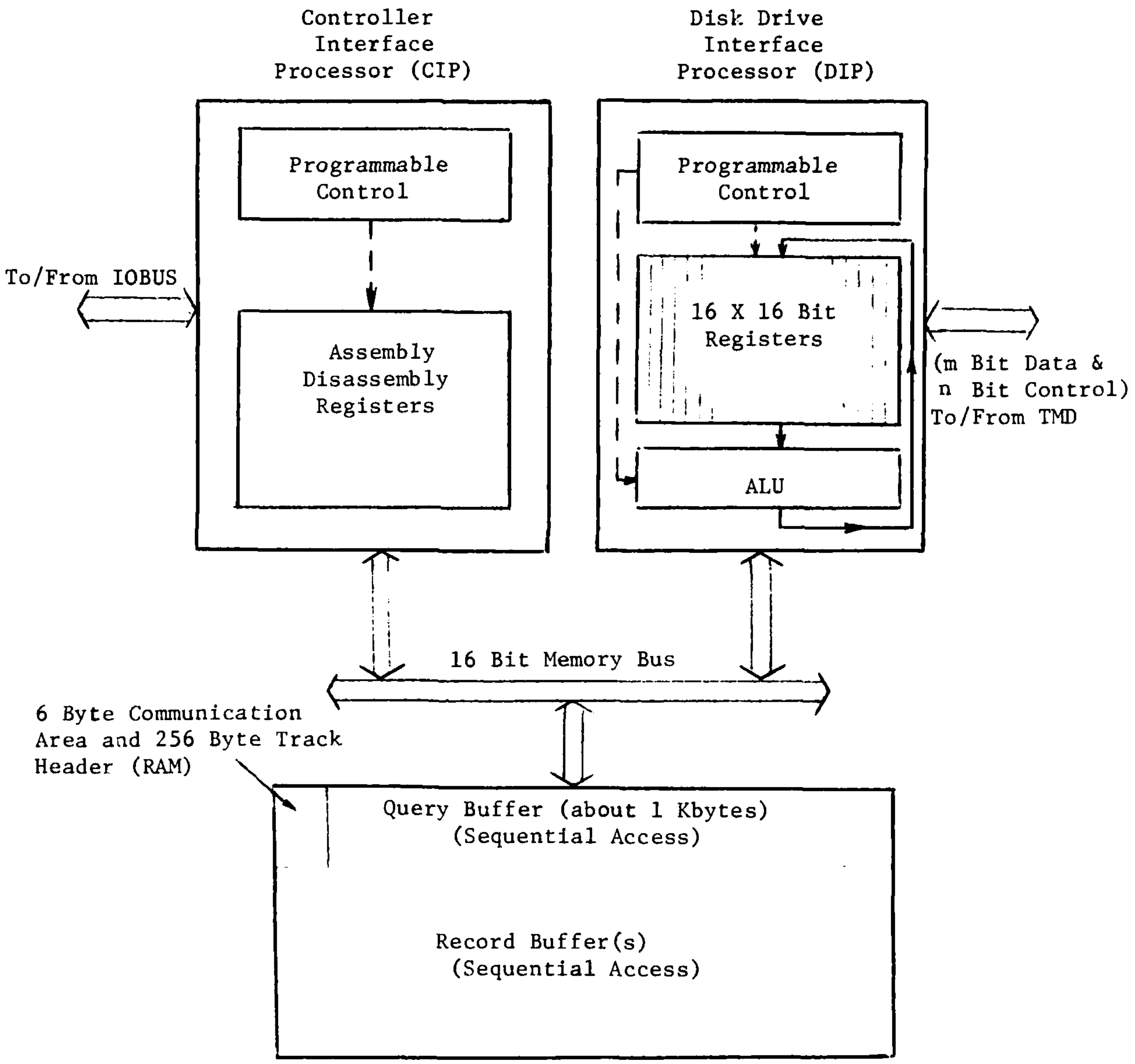

Controller Interface Processor (CIP)
Disk Drive Interface Processor (DIP)
Programmable Control
Assembly Disassembly Registers
To/From IOBUS
Programmable Control
16 X 16 Bit Registers
(m Bit Data & n Bit Control) To/From TMD
ALU
16 Bit Memory Bus
6 Byte Communication Area and 256 Byte Track Header (RAM)
Query Buffer (about 1 Kbytes) (Sequential Access)
Record Buffer(s) (Sequential Access)

Figure 10. Organization of a Track Information Processor

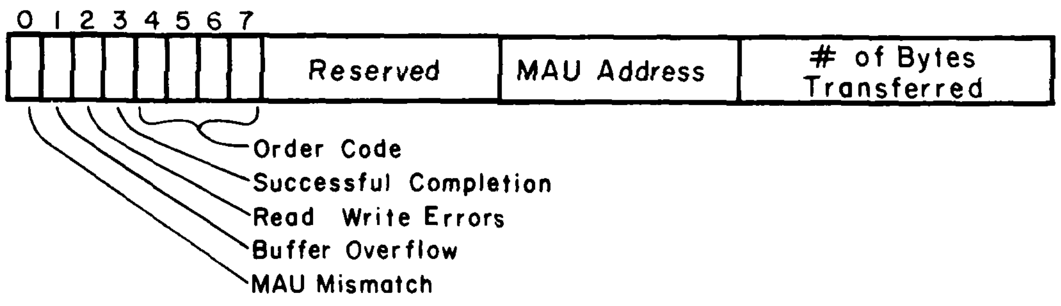
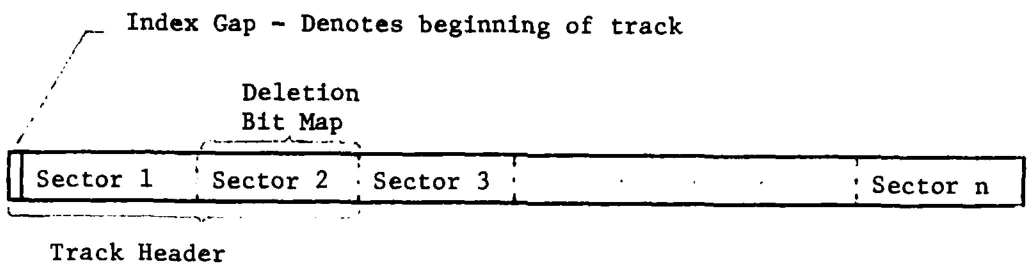
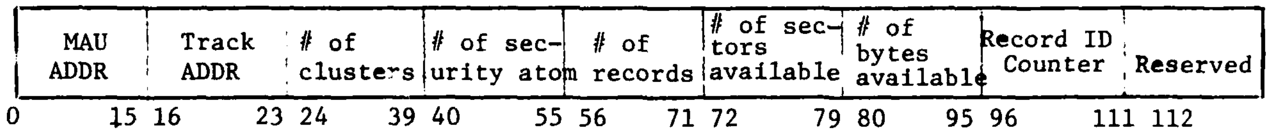


Figure 11. Format of the Communication Area between the CIP and the DIP



Track Format



Format of the First Sector on Track

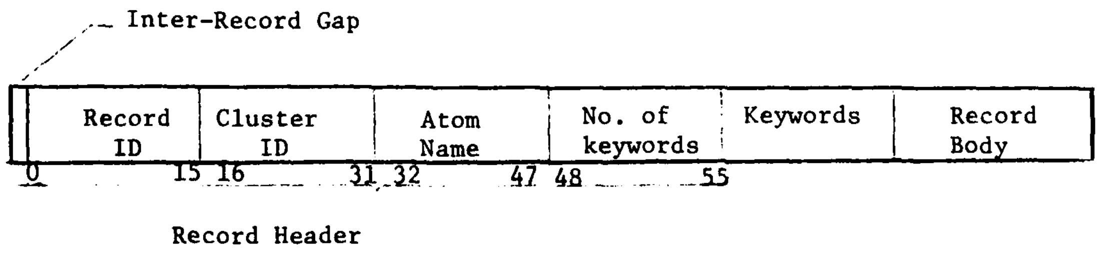


Figure 12. Format of a Track as Perceived by a TIP

The controller interface processor (CIP) can be a commerciallv available microprocessor capable of transferring information from the record buffer to the MMC via the IOBUS.

3. INSERION OF RECORDS IN DBC

This section will be devoted to an explanation of the process of insertion of records into the mass memory (MM) of DBC. Also, certain architectural elements are proposed to respond more quickly to insertion requests. However, the proposed elements are simple and do not require any major technological breakthrough. The advantages that will accrue as a result of the proposal are analyzed the last sub-section. These elements will become increasingly attractive in an update-intensive environment - an environment in which a very large proportion of the requests to DBC will consist of inserts, deletes, and updates.

3.1 Motivation for the Proposed Elements

A simulation study [15] of DBC was conducted in order to determine the potential bottlenecks to its smooth performance. It was discovered that the data loop may not be able to match the throughput of the structure loop. The study showed that the data loop should have a throughput of about 35 orders per second in order to be compatible with the throughput of the structure loop. However, it was discovered that the MM design allowed only for a throughput of 20 orders per second. One of the suggestions made as a result of that study [15] was that the data loop be speeded up. Later on, in this section, we shall propose a means for improving the throughput of the data loop.

3.2 Insert-in-Parallel

Earlier, in Section 2, we had made the observation that our design of MM allowed for a parallel transfer of data from the track information processors (TIPs) to the tracks of the moving-head disks (besides allowing for the parallel transfer in the opposite direction from the tracks to the TIPs which we have termed tracks-in-parallel-readout). We propose to make use of this parallel-write-in facility to improve the processing speed of the data loop.

Let us recapitulate the process involved in inserting a record into the MM of DBC. First, the database command and control processor (DBCCP) determines the security atom and cluster that the record belongs to [7]. Next, it determines the cylinder number (MAU) into which the record should be inserted. Finally, the structure memory (SM) is accessed and one entry is made in it per directory keyword in the record. That is, for each directory keyword in the record, an index entry indicating the security atom of the record and the cylinder in which it is to be inserted is created in SM. The record is now presented to the data loop which then proceeds to insert it into the MAU selected by DBCCP.

The mass memory monitor (MMM) determines the track within the cylinder into which the record is to be inserted by querying all the TIPs. We recall, from Figure 12, that this information is available in the first sector of each track. The record to be inserted is then sent form the MMM, via the IOBUS, to be placed in the TIP buffer corresponding to the selected track (remember that there is a one-to-one correspondence between tracks in a cylinder and the TIPs). The MMM now issues the'insert-record' request to the selected TIP, again using the IOBUS. The TIP does the insertion in one revolution of the disk device. We note that during the course of this revolution, only one out of all the TIPs (20-40) is doing useful work. This is a utilization ratio of between 1/20 and 1/40. In the following paragraphs, we outline a method for improving the utilization of the TIPs.

There are two possible schemes to take advantage of the inherent parallelism present in the MM architecture. These schemes are merely two different ways of implementing the same logical idea. We shall call them Scheme 1 and Scheme 2, respectively.

The basic idea is to insert many records in parallel, all in the same revolution. How many records can be inserted in parallel in one disk revolution? Theoretically, as many records as can fit into one cylinder may be inserted in one revolution of the disk device.

We now describe the first of the two schemes. The operation proceeds as follows. First, as each record is presented for insertion to DBC, the <u>cylinder and the track within the cylinder</u> into which it must be inserted is determined by DBCCP. All records that are to be inserted into the same cylinder are grouped together and only one 'insert-records' request is issued for this entire group of records. When the mass memory monitor (MMM) comes around to executing this order, each record is first placed in the buffer of the TIP corresponding to the track in which the record is to be inserted. This communication between the MMM and the TIPs is conducted via the IOBUS. The 'insert-records' request is then broadcast to all the TIPs using the IOBUS. AS many records per track may be inserted in a disk revolution as the size of the TIP buffer will allow. If the TIP buffer is as big as the size of a track, then an entire cylinder's worth of records may be inserted per disk revolution.

The problem with the above method is that just before the TIPs begin to execute the 'insert-records' request, the IOBUS will be congested by traffic owing to the large number of records that must be sent to the TIPs for insertion. This may cause a delay of one or more disk revolutions in addition to the disk revolution needed to insert all the records. A way to avoid this delay is to have direct connections from the MMM to each of the TIPs and to do away with the IOBUS. Figure 13 illustrates the situation. An alternative scheme that does not need these costly additional communication lines and yet avoids most of the delay of Scheme 1 is proposed below.

Once again, DBCCP first determines, for each record to be inserted, the <u>cylinder and the track</u> into which the record is to be inserted. However, no grouping of records that are to be inserted into the same cylinder is done. Instead, the DBCCP sends out these records for insertion, one at a time. Each 'insert-record' order, as received by the MMM, has two arguments - the record to be inserted and the cylinder and track in which it must be inserted. When the MMM has to insert a record, it places the record in the buffer of the TIP corresponding to the track chosen for inserting the record. This may be done by the MMM when the TIPs are doing other useful work. For example, after the MMM issues a 'delete-by-query' request, it waits for the TIPs to delete those records that satisfy the given query. At the end of the deletion process, the TIPs will interrupt the MMM. During the time that the TIPs are busy performing the deletion (that is, for one revolution of the disk device), microsequencer MC-2 of the MMM (see Section 2) is idle. This idle time of the MMM may be fruitfully employed in order to place records for insertion into TIP buffers. Similarly, the MMM is idle after it issues an update request until the time it is interrupted by the TIPs (to indicate that the TIPs have furnished processing the update request). However, the time between the issuance of a 'retrieve-by-query' request to the TIPs by the MMM and the receipt of an interrupt by the MMM from the TIPs (indicating that the TIPs have completed processing the request) may <u>not</u> be utilized to place records in the TIP buffers because the retrieved records are being sent to the MMM via the IOBUS.

After the record to be inserted has been placed in the appropriate TIP buffer, the MMM continues the processing of other requests. The above logic is repeated for every 'insert-record' request. That is, the MMM places the record in the next available space in the TIP buffer corresponding to the track chosen for insertion. After all the requests on a particular MAU have been

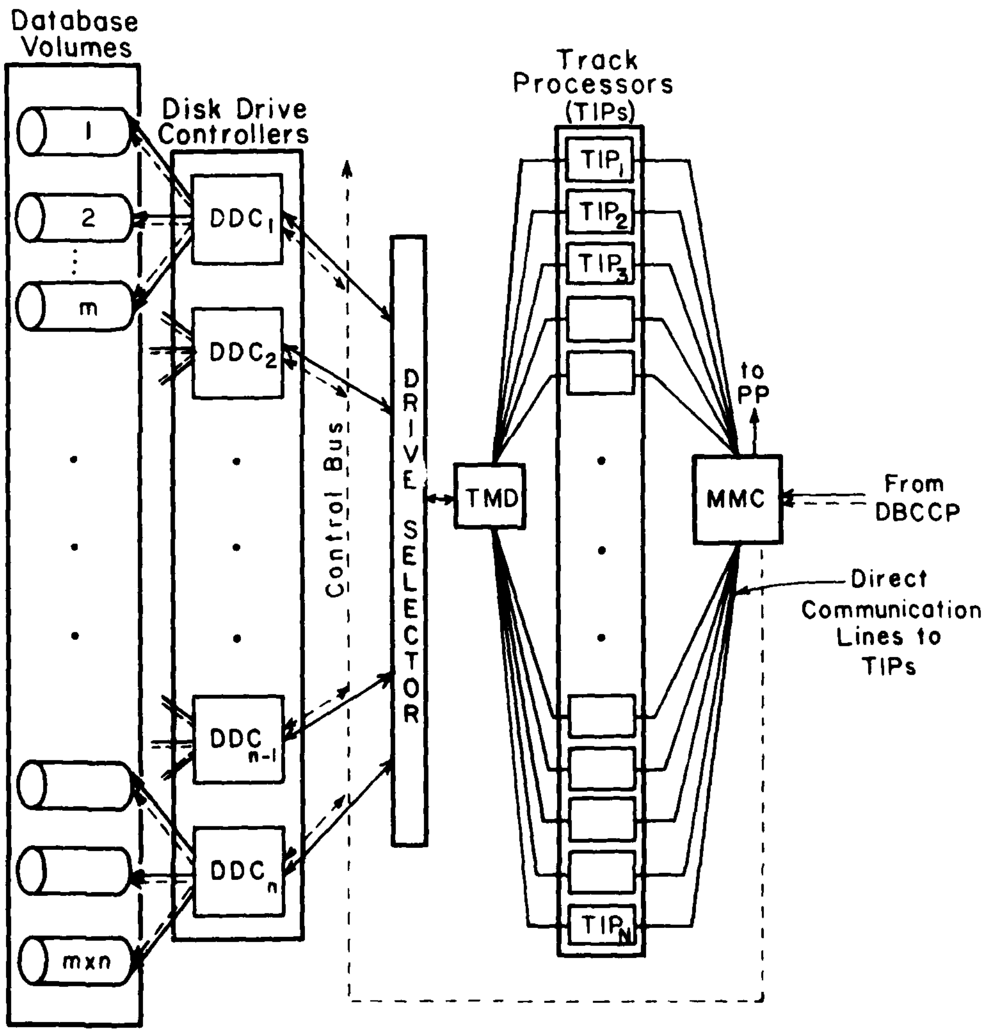


Figure 13. A New Bus Structure for the Mass Memory

completed (i.e., after the execution of a set of orders), one additional revolution is used to insert the records present in the TIP buffers. We may recall, from Section 2, that one additional revolution is required at the end of each set of orders (i.e., after all the orders on a MAU have been completed and before a set of orders on another MAU is chosen for execution) in order to rewrite the bit map onto the beginning of each track in the cylinder. This same revolution of the disk device may be utilized to insert all the records present in the TIP buffers. Thus, all the insert record requests present in one set of orders can be executed without incurring a single extra disk revolution. For example, if the average set of orders has ten insert commands, then the proposed scheme will result in a saving of ten revolutions of the disk device per set of orders. This will considerably improve the performance and throughput of the data loop.

3.3 Comparison of the Two Methods

Scheme 2 has a disadvantage which is not present in Scheme 1 because it demands that each TIP have two buffers. This is because the TIP buffers need to be used during the execution of retrieve and update queries in order to store records to be retrieved and sent to the security filter processor (SFP) or to store records to be updated. Hence, an additional buffer will need to be used to store the records for insertion since, in Scheme 2, many retrieve and update requests may be executed in the time between the placement of a record for insertion into a TIP buffer and the actual insertion of that record onto the track. The size of the second buffer will depend upon such factors as the number of 'insert-record' requests that are expected per set of orders, the distribution of these requests among the tracks and the size of the records. However, this buffer need be no larger than the size of a track. We may recall that the TIP record buffer consists of a number of individually accessible segments. A number of these segments may be set aside for use as the second buffer. For example, if the buffer consists of M segments, Segments 1 through m may be used as the second buffer to hold records for insertion. Segments m+1 through M will then be used by the TIPs only for retrieval and update purposes. We have already mentioned the considerations to be made before arriving at a decision for the value of m.

The two methods differ only in the way in which the records are placed in the TIP buffers. Scheme 1 waits until both the MMM and the TIPs are idle before placing the records in the buffers. Scheme 2 can utilize moments when the TIPs are busy and the MMM is idle. Since the transfer of records from the MMM to the TIP buffers does not require the participation of the TIPs, this latter scheme is entirely feasible. However, at this point, we would like to add a note of caution. Consider what happens when a file is being loaded into the MM of DBC. It is possible that a large number of 'insert-record' requests occur in a sequence, one after the other. Scheme 2 takes advantage of the presence of requests other than 'insert-record' requests in order to overlap useful TIP processing time on these other requests with MMM processing time on 'insert-record' requests. However, if a large number of 'insert-record' requests occur in sequence, this overlap may not be attainable to the extent desired. In the worst case, when all the requests in a particular set of orders are 'insert-record' requests, Scheme 2 will take as long as Scheme 1 to execute.

3.4 Algorithm Executed by MMM

The algorithm that will be executed by the MMM in order to process a set of orders on a MAU on which a seek has already taken place is described below. It is a modification of Algorithm B of Appendix 1. The algorithm is the one that will be employed if Scheme 2 is implemented.

```
ALGORITHM B MODIFIED: To initiate the execution of orders by the TIPs and to
accept data retrieved by the TIPs.

Input Arguments: 1. The number of N of orders pending execution.
                 2. The address of the first order in the order table (OT).

Step 1:   [Initialize] p=1.  FLAG=Ø.
Step 2:   Pick up the pth. order from the OT. If the order code indicates
          an insert-record order, go to Step 6. If the order code indicates
          a delete-record order, then go to Step 5. If the order code indicates
          an update order, then go to Step 7. If the order code indicates a
          compaction order, then go to Step 15.
Step 3:   [Retrieve] Broadcast the order to all the TIPs and go to Step 14.
Step 4:   [Here we try to utilize MMM idle time to place records for insertion
          into TIP buffers] If FLAG=Ø, then continue execution of Algorithm
          C (which places records for insertion into TIP buffers) until TIP
          interrupt occurs. If FLAG=1, then wait until TIP interrupt occurs.
          When the interrupt occurs, go to Step 8.
Step 5:   [Delete] Broadcast the order to all the TIPs. Turn on DELETE flag.
          Go to Step 4.
Step 6:   [Insert] Check the mark bit to see if the order has been taken care of
          by Algorithm C. If so, then go to Step 13. Else, place the record
          to be inserted into the buffer of the TIP corresponding to the
          track chosen for insertion. Go to Step 13.
Step 7:   [Update] Broadcast the order to all the TIPs. Turn on the
          UPDATE flag. Go to Step 4.

INTERRUPT ENTRY
Step 8:   If the UPDATE flag is on, go to Step 9. If the DELETE flag is on,
          then go to Step 11.
```

Step 9: [This part of the algorithm will be described in Section 5.]
Step 11: [Check if there was any deletion.] Turn off the DELETION flag. If the TIPs indicate that some records were tagged for deletion, then go to Step 12, else go to Step 13.
Step 12: Store the MAU address in the mass memory deletion table (MMDT).
Step 13: Delete the order from the OT. p=p+1. If p>N, then request the TIPs to write back all deletion tags and to insert all the records in their buffers into the tracks, set IDLE flag on and halt; else go to Step 2.
Step 14: [Receive retrieved records] If the TIPs have records to be output, then receive them and send them to the SFP. Go to Step 13.
Step 15: [Compaction] The algorithm for compaction will be discussed in the next section.

ALGORITHM C: This algorithm is executed by the MMM whenever it is waiting for a TIP interrupt. The interrupt causes the abandonment of the execution of this algorithm. The point of interrupt is remembered, and the algorithm is resumed at a later time by the MMM when it is idle waiting for a TIP interrupt.

Step 1: q=q+1. If q>N, then set FLAG=1 and terminate.
Step 2: Look at the qth order in the OT. If it is an 'insert-record' order, then set the mark bit corresponding to this order and go to Step 3. Else, go to Step 1.
Step 3: Place the record to be inserted into the buffer of the TIP corresponding to the track chosen for insertion. Go to Step 1.

3.5 Determining the Track for Placement of a Record

In this sub-section, we shall demonstrate that it is indeed possible for DBCCP to determine, for each record submitted for placement, the cylinder and the track within that cylinder in which the record must be stored. However, before we explain the method adopted in DBC to determine the cylinder and track for insertion of a record, the concept of clustering must be well understood. Accordingly, the first sub-section below describes the concepts behind the strategies employed in DBC to place data elements that have a high probability of being retrieved and updated together, in close proximity of one another. Certain data structures used by the algorithm are then presented, followed by the algorithm itself.

We would like to mention, in passing, that the determination of the track for insertion may be done by the TIPs since information regarding space availability is part of the track header information in each track (Figure 12). However, we must keep in mind the results of the simulation study [15]. That is, the throughput of the data loop is lesser than that of the structure loop. Thus, whenever a piece of work can be performed either in the data loop or the structure loop, we shall choose to do it in the structure loop in order to close up the difference in throughput rates of the two loops.

3.5.1 Clustering Descriptors and Logical Clusters

A file is associated with a single primary clustering attribute and any number of secondary clustering attributes. The latter attributes are specified in an order of importance. The importance of a secondary clustering attribute is defined to be its relative position in the above list. Thus, we can talk of one secondary clustering attribute as being more important than another secondary clustering attribute for clustering purposes.

At the time of file creation, the file creator also specifies a set of clustering descriptors. These descriptors may be of one of three types:

Type A: The descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to-predicate, such that the same attribute appears in both predicates. An example of a type-A descriptor is as follows:

$$((\text{Salary} \geq 2000) \wedge (\text{Salary} \leq 10000)).$$

More simply, this is written as follows:

$$(2000 \leq \text{Salary} \leq 10000).$$

Thus, the file creator merely specifies an attribute (i.e., Salary) and a range of values (2000 - 10000) for that attribute.

Type B: The descriptor is an equality predicate. An example of a type-B descriptor is:

(Position=PROFESSOR).

Type C: The descriptor consists of only an attribute name. Let us assume that there are n different keywords K1, K2, ..., Kn, in the records of this file, with this attribute. Then, this type-C descriptor is really equivalent to n type-B descriptors B1, B2, ..., Bn, where Bi is the equality predicate satisfied by Ki. In fact, this type-C descriptor will cause n different type-B descriptors to be formed. These type-B descriptors formed from a type-C descriptor are known as type-C sub-descriptors.

The attribute that appears in a clustering descriptor must be either the primary clustering attribute or one of the secondary clustering attributes. A clustering descriptor is a primary (secondary) clustering descriptor if the corresponding attribute is a primary (secondary) clustering attribute.

A primary (secondary) clustering keyword is a keyword of a record such that one of the following holds:

(a) The attribute of the keyword is specified in a type-A primary (secondary) clustering descriptor and the value is within the range of that descriptor.

(b) The attribute and value of the keyword match those specified in a type-B primary (secondary) clustering descriptor.

(c) The attribute of the keyword is specified in a type-C primary (secondary) clustering descriptor.

In all these cases, the primary (secondary) clustering keyword is said to be derived or derivable from the corresponding primary (secondary) clustering descriptor. Each primary clustering descriptor is associated with a maximum space requirement (in terms of number of cylinders) which indicates the estimated amount of storage required in the mass memory for all records having keywords derived from this descriptor. The importance of a secondary clustering keyword is defined to be the same as the importance of the corresponding secondary clustering attribute of the keyword.

Each (primary clustering descriptor, secondary clustering descriptor) pair defines a cluster of records. Each record in this cluster must satisfy two conditions.

(1) The primary clustering keyword of the record should be derivable from the primary clustering descriptor of the cluster.

(2) The most important secondary clustering keyword of the record should be derivable from the secondary clustering descriptor of the cluster.

A record for insertion must have a primary clustering keyword; otherwise, the record will be rejected by DBC. If a record has no secondary clustering keyword, then a null secondary clustering keyword derivable from a null secondary clustering descriptor is assumed. The null secondary clustering keyword has least importance.

We illustrate these concepts by means of an example developed in Figures 14a, 14b, 14c and 14d. Figure 14a shows a file consisting of six records, where each record has four attributes. As can be seen from the figure, the user has specified the primary clustering attribute to be Job and the secondary clustering attributes in order of importance to be Salary and Department Number. In Figures 14b and 14c, the primary and secondary clustering descriptors of the file have been shown. Finally, in Figure 14d, we illustrate the various clusters formed for the file and their composition both in terms of primary

PRIMARY CLUSTERING ATTRIBUTE: JOB
SECONDARY CLUSTERING ATTRIBUTES IN ORDER OF
IMPORTANCE: SALARY, DAPARTMENT-NUMBER

RECORD NO.	NAME	JOB	SALARY	DEPARTMENT-NUMBER
1.	HAYES	MANAGER	1500	100
2.	NAYAK	ENGINEER	2000	100
3.	BOONE	TECHNICIAN	4000	200
4.	WHITE	MANAGER	1200	700
5.	KLINE	ENGINEER	2500	200
6.	PRICE	ENGINEER	2500	200

Figure 14a. The Six Records of a File, and its Primary and Secondary Clustering Attributes

DESCRIPTOR ID.	DESCRIPTOR	TYPE
PY1.	JOB = MANAGER	B
PY2.	JOB = ENGINEER	B
PY3.	JOB = TECHNICIAN	B

Figure 14b. Primary Clustering Descriptors of File F

DESCRIPTOR ID.	DESCRIPTOR	TYPE
SEC1.	$1000 \leq$ SALARY ≤ 2000	A
SEC2.	$2001 \leq$ SALARY ≤ 3000	A
SEC3.	DEPT. NUMBER = 100	B
SEC4.	DEPT. NUMBER = 200	B

Figure 14c. Secondary Clustering Descriptors of File F

CLUSTER NUMBER	PRIMARY DESCRIPTOR	MOST IMPORTANT SECONDARY DESCRIPTOR FOR THE RECORD	RECORDS IN CLUSTER
1	PY1	SEC1	R1, R4
2	PY2	SEC1	R2
3	PY3	SEC4	R3
4	PY2	SEC2	R5, R6

Figure 14d. The Clusters of File F and their Composition

and secondary clustering descriptors, and in terms of the records that make up the clusters. For example, the record R1 belongs to the cluster 1, which is defined by the primary clustering descriptor PY1 and the secondary clustering descriptor SEC1. To see why this is so, let us look at the keywords of R1. R1 contains the keyword <Job, MANAGER> and this is obviously derivable from the primary clustering descriptor PY1, i.e., (Job=MANAGER). Also, R1 contains the keyword <Salary, 15ØØ> and this is derivable from the secondary clustering descriptor, SEC1, (i.e., 1ØØØ≤Salary≤2ØØØ) since, 15ØØ lies between 1ØØØ and 2ØØØ. It also contains the keyword <Department Number, 1ØØ>, which is derivable from another secondary clustering descriptor, SEC3. However, for clustering purposes, we only consider the primary clustering descriptor PY1 and the secondary clustering descriptor SEC1 of R1 from which the primary clustering keyword of the record (i.e., <Job, MANAGER>) and the most important secondary clustering keyword of the record (i.e., <Salary, 1500>) are derivable. Hence, R1 belongs to the cluster defined by PY1 and SEC1.

3.5.2 Data Structures Used in the Algorithm

A. The Track Space Table (TST)

The track space table (TST) is used by DBCCP to determine, for each record for insertion, the cylinder and track into which it must be inserted. A logical view of this table is shown in Figure 15.

There is one such table for every file known to DBC. There are as many entries in the TST as there are cylinders allocated to the file. We may recall [7], that a file is allocated a certain number of cylinders and that a cylinder only contains records of a single file. Each entry consists of N+1 fields, where N is the number of tracks in a cylinder (N is in the range 20 - 40). The first field contains a cylinder number. The i-th field (2<i<N+1) contains the number of bytes available for allocation in track i-1 of the cylinder contained in Field 1.

B. The Cluster Identifier Definition Table (CIDT)

There is one such table for every file known to DBC. Each entry in the table is a quadruple as shown in Figure 16. The first field contains a cluster identifier. The second and third fields contain the identity of the primary clustering descriptor and the secondary clustering descriptor, respectively, that make up this cluster. The fourth field contains the cylinder number of a cylinder that contains at least one record belonging to this cluster.

Cylinder Number	Space Available in Track 1	Space Available in Track 2	• • •	Space Available in Track N
		•		
		•		
		•		
			• • •	

N is the number of tracks in a cylinder.

Figure 15. The Track Space Table (TST)

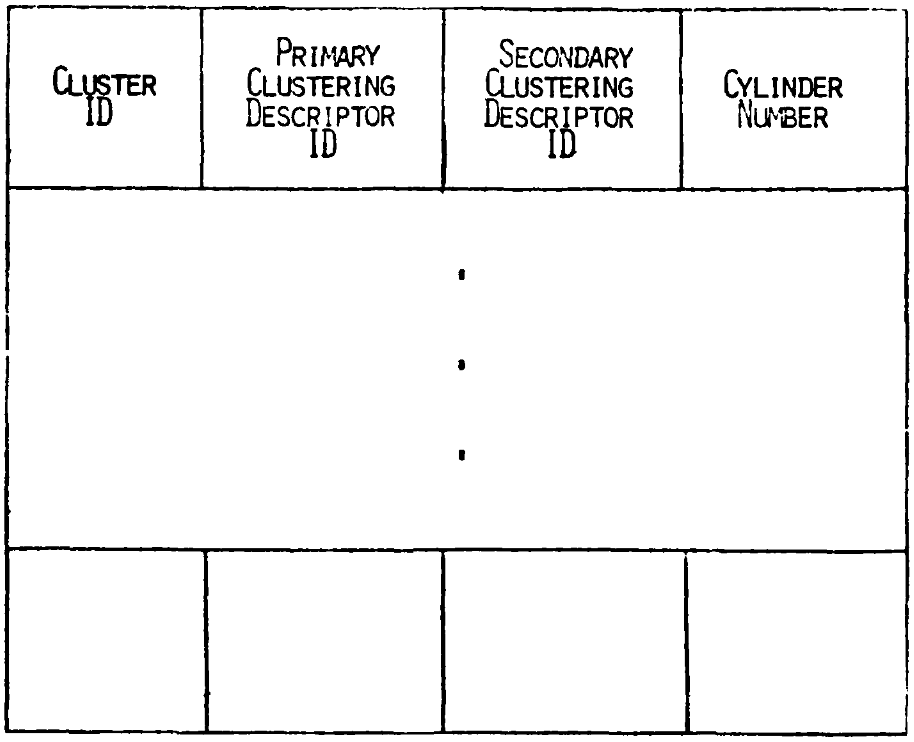

CLUSTER ID
PRIMARY CLUSTERING DESCRIPTOR ID
SECONDARY CLUSTERING DESCRIPTOR ID
CYLINDER NUMBER

Figure 16. The Cluster Identifier Definition Table (CIDT)

C. The Primary Clustering Descriptor Table (PCDT)

The logical view of one such table is shown in Figure 17. There is one primary clustering descriptor table (PCDT) for each file known to DBC. There is an entry in the PCDT for each primary clustering descriptor and each entry consists of five fields as shown in the figure. The first field contains the identifier of the attribute in the primary clustering descriptor. In case the descriptor is of type-A, then the second and third fields contain the upper and lower limits, respectively, of the range that defines the descriptor. If the primary clustering descriptor of sub-descriptor is of type-B or type-C, then the second field is null and the third field stores the value of the descriptor or sub-descriptor. The fourth field contains the identifier assigned to the descriptor by DBC. The fifth field contains the maximum space requirement for the clustering descriptor. This maximum space requirement is the estimated number of cylinders needed for storing all records whose primary clustering keywords are derivable from the clustering descriptor.

3.5.3 The Algorithm

There are two inputs to this algorithm. These are the primary clustering descriptor LIST[1] and the secondary clustering descriptor LIST[2] from which the primary clustering keyword and the most important secondary clustering keyword, respectively, of the record are derived. For an explanation of how LIST[1] and LIST[2] are obtained, the reader is referred to [7].

ALGORITHM: To select a cylinder and a track within that cylinder into which a given record may be inserted.

Step 1: Set FLAG=1. Search the CIDT of the file to which the record belongs, looking for all entries whose second field contains LIST[1] and whose third field contains LIST[2]. Extract the fourth field from all such entries and put them in a set CYL. If CYL contains no elements, then go to Step 7. Else, go to Step 2.

Step 2: For every member of CYL, do Step 3.

Step 3: Search the TST of the file to which the record belongs, looking for an entry whose first field matches the element of CYL. Place the entire matching entry (which is an N+1 tuple) into a set h.

Step 4: Let the i-th member of h be denoted by <Ci, T1i, T2i, ..., TNi>. For each member of h, find the largest element among the entries T1i, ..., TNi. Denote the maximum in the i th element of h by T(ki)i. [That is, let it be the entry in the (ki+1)-th field.] Let Tmax be the maximum among all T(ki)i. [In choosing maxima, if more than one exist, choose one arbitrarily.] Let Tmax=T(kj)j. [That is, let it be the entry in the (kj+1)-th field of the j-th member of h.]

ATTRIBUTE ID	LOWER LIMIT	UPPER LIMIT	PRIMARY CLUSTERING DESCRIPTOR ID	MAXIMUM SPACE REQUIREMENT
		.		
		.		
		.		
		.		

Figure 17. The Primary Clustering Descriptor Table (PCDT)

Step 5: Compare Tmax with the length of the record (lr) to be inserted. If Tmax≥lr, then change the TST entry from <Cj, T1j, T2j, ..., T(kj)j, ..., TNj> to <Cj, T1j, T2j, ..., T(kj)j-lr, ..., TNj> and insert the record in track kj of cylinder j and terminate. If Tmax<lr and Flag=1, then go to Step 6. If Tmax<lr and FLAG=2, then go to Step 7. If Tmax<lr and FLAG=3, then terminate with a negative signal.

Step 6: Search the CIDT, looking for all entries whose second field contains LIST[1]. Extract the fourth field from all such entries and put them together in a set CYL. Compare the number of unique elements in CYL (|CYL|) with the estimated number of cylinders br needed for storing all records whose primary clustering keyword is derivable from the primary clustering descriptor in LIST [1]. This number may be found from the fifth field of the entry in the PCDT, whose fourth field matches LIST[1]. If |CYL|<br, then go to Step 7. Else, set FLAG=2 and go to Step 2.

Step 7: Set FLAG=3. Put all the entries in the TST into a set h. Go to Step 4.

3.6 Analytical Study of Data Loop Performance

In this final sub-section, we propose to conduct a study into the performance of the data loop, in order to show, in concrete terms, the throughput improvement that will be caused by the proposed design. We begin by describing the model used for the analytical study.

3.6.1 The Model

In Figure 18, we have shown the model used for analysis of the data loop. User requests, which have been pre-processed by the structure loop or which do not require pre-processing, are placed in one of the m disk drive controller (DDC) queues. There is one such queue per disk drive controller. (Note that each of the disk drive controller queues really consists of p*q cylinder queues, where p is the number (8-16) of disks under the control of a disk drive controller, and q is the number (200-400) of cylinders per disk drive.)

Certain simplifying assumptions are now made in order to make the solution tractable. First, for a given disk drive, we assume that a DDC cannot issue more than one seek at a time. That is, a DDC issues a second seek only after the set of orders on the cylinder referred to in the first seek have been executed by the TIPs. Also, we assume an exponentially distributed arrival rate of orders from the structure memory (SM). Additionally, it is hoped that these orders are going to be evenly distributed among the various cylinders in the MM. Finally, we assume that the mass memory monitor (MMM) adopts a round-robin policy in dealing with requests. That is, it begins by interrogating the first DDC to see if it has any completed seek. If so, it initiates

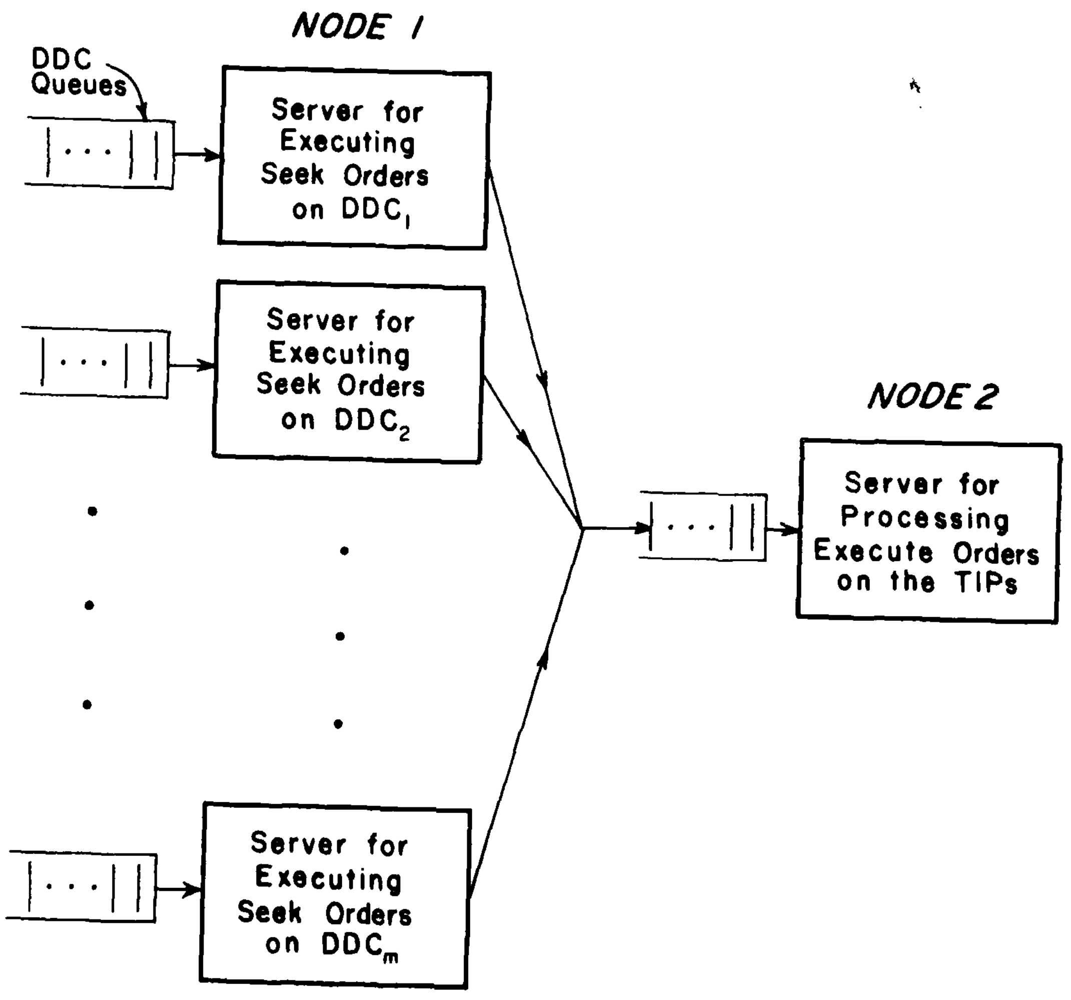

NODE 1
DDC Queues
Server for Executing Seek Orders on DDC$_1$
Server for Executing Seek Orders on DDC$_2$
Server for Executing Seek Orders on DDC$_m$
NODE 2
Server for Processing Execute Orders on the TIPs

Figure 18. The Model Used for the Analytical Study

execution of the set of orders for the cylinder on which the seek is complete. Else, it looks at the second DDC, and so on.

We note that the model adopted is a two-node network model, where the first node consists of m exponential servers (m is the number of DDCs) and the second node has one exponential server. The problem is complicated by the fact that the i-th server in the first node is blocked [16] from initiating a second seek on a particular cylinder until the server in the second stage can process the set of orders on the cylinder for which the i-th server has completed a seek. The following paragraphs outline the method employed to overcome this complication that exists between the two nodes of the network model.

Let us assume that an order from the DBCCP arrives every n milliseconds. With m DDCs, and a uniform distribution of requests among the cylinders, this means that an order is placed in one of the DDC queues, one every m*n milliseconds. Assuming p disks to be under the control of a DDC and q cylinders per disk, an order is placed in one of the cylinder queues, one every m*n*p*q milliseconds. Let us also assume that a seek is not issued on a cylinder until at least k orders exist for execution in the cylinder queue of that cylinder (this is a design decision). The time taken before a queue has k orders is k*m*n*p*q milliseconds. But, in k*m*n*p*q milliseconds, all p*q cylinders of a DDC have k orders waiting for execution (since the orders will be evenly distributed among the cylinders). Thus, seek orders are generated for each of the m servers in the first node at an average rate of (k*m*n*p*q)/(p*q) = k*m*n milliseconds. This is the inter-arrival rate of orders at each of the m servers in the first node. Assuming a Poisson arrival distribution, we have, as the arrival distribution, the following formual:

$$P(t) = \frac{1}{kmn} * e^{-\frac{t}{kmn}}$$

Let us now proceed to calculate the time taken by the server in the second node to process a request consisting of k orders Let

p1	=	percentage of retrieve requests
t1	=	time taken to execute a retrieve request by the TIPs
p2	=	percentage of update requests
t2	=	time taken to execute an update request
p3	=	percentage of delete requests
t3	=	time taken to execute a delete request
p4	=	percentage of insert requests
t4	=	time taken to execute an insert request

Let T(k) = time taken to execute a set of k orders (by the TIPs). Then,

$$T(k) = k * [p1*t1 + p2*t2 + p3*t3 + p4*t4]$$

The average time taken to execute a set of orders Tav, depends upon L, the average size of a set of orders. This will be greater than k (the number of orders needed per set of orders before a seek is initiated) since, during the time between the initiation of a seek on a cylinder and the execution of the set of orders on that cylinder, more orders on that cylinder could have arrived. In the next paragraph, we shall try to calculate L.

Let us first adopt the following notation. We shall number the DDCs as DDC1, DDC2, ..., DDCm. Also, we donate the j-th disk drive of DDCi as Dij. Recall that the TIP scheduling policy is round-robin. That is, a set of orders on some cylinder of D11 is first executed. Next, a set of orders on some cylinder of D12 is executed and, st the same time, DDC1 initiates a seek on some cylinder of D11. The TIPs now execute a set of orders on a cylinder of D13, then a set of orders on a cylinder of D14, and so on. Thus, we see, that between the initiation of a seek on a cylinder of a particular disk drive and the execution of a set of orders on that cylinder, the time elapsed is exactly equal to the time spent by the TIPs in executing one set of orders from each of the other disk drives. (Assuming that at least one seek has been completed on each of the other disk drives. This is a valid assumption, if L is large enough that the time taken by the TIPs in executing a set of orders of size L is greater than the average seek time of the disk mechanism.)
Time taken by the TIPs to execute mp-1 orders each of average size L

$$= (mp-1) * L * [p1*t1 + p2*t2 + p3*t3 + p4*t4]$$
$$= (mp-1) * Tav$$

In this time, the size of the cylinder queues have grown from k to L. But, new arrivals to a cylinder queue come with an inter-arrival rate of m*n*p*q milliseconds. Thus,

$$(L-k)*m*n*p*q = (mp-1) * L * [p1*t1 + p2*t2 + p3*t3 + p4*t4].$$

This gives us

$$L = \frac{k*m*n*p*q}{m*n*p*q - (mp-1) * [p1*t1 + p2*t2 + p3*t3 + p4*t4]}$$

and

$$Tav = L * [p1*t1 + p2*t2 + p3*t3 + p4*t4]$$

Tav is the average time taken by the server in the second node to execute a set of orders. Assuming an exponentially distributed service time, we have the following distribution for the server in the second node.

$$\text{Pserver2}(t) = \frac{1}{Tav} e^{-\frac{t}{Tav}}$$

We are now in a position to calculate the average time taken by a server in the first node. It is obvious, from the discussion of the previous paragraph, that each DDC issues p seeks in a time equal to the average time taken by the TIPs to execute mp sets of orders. Thus, the average service rate of servers in the first node is m*Tav. Again, assuming an exponential distribution, the distribution of the service time for a server in the first node is:

$$\text{Pserver1}(t) = \frac{1}{m*Tav} e^{-\frac{t}{m*Tav}}$$

The model is now as shown in Figure 19, It is a network of two nodes, the first node consisting of m servers with exponentially distributed service times and exponential arrival rates, and the second node consisting of a single server with an exponentially distributed service time. All the distributions are known, and the solution to the above problem is well known.

3.6.2 The Results of the Study

The problem is now borken up into two parts as shown in Figures 20a and 20b. Each part consists of an M/M/1 queue [17]. Note that the second part consists of a single server with an average arrival rate which is m times the average arrival rate of the first part. We will now try to find out the conditions under which the queues will be steady, i.e., the queues do not build up indefinitely. We know that, in each queue, the ratio of the average arrival rate to the average service rate must be less than one for stability [17].
From Figure 20a,

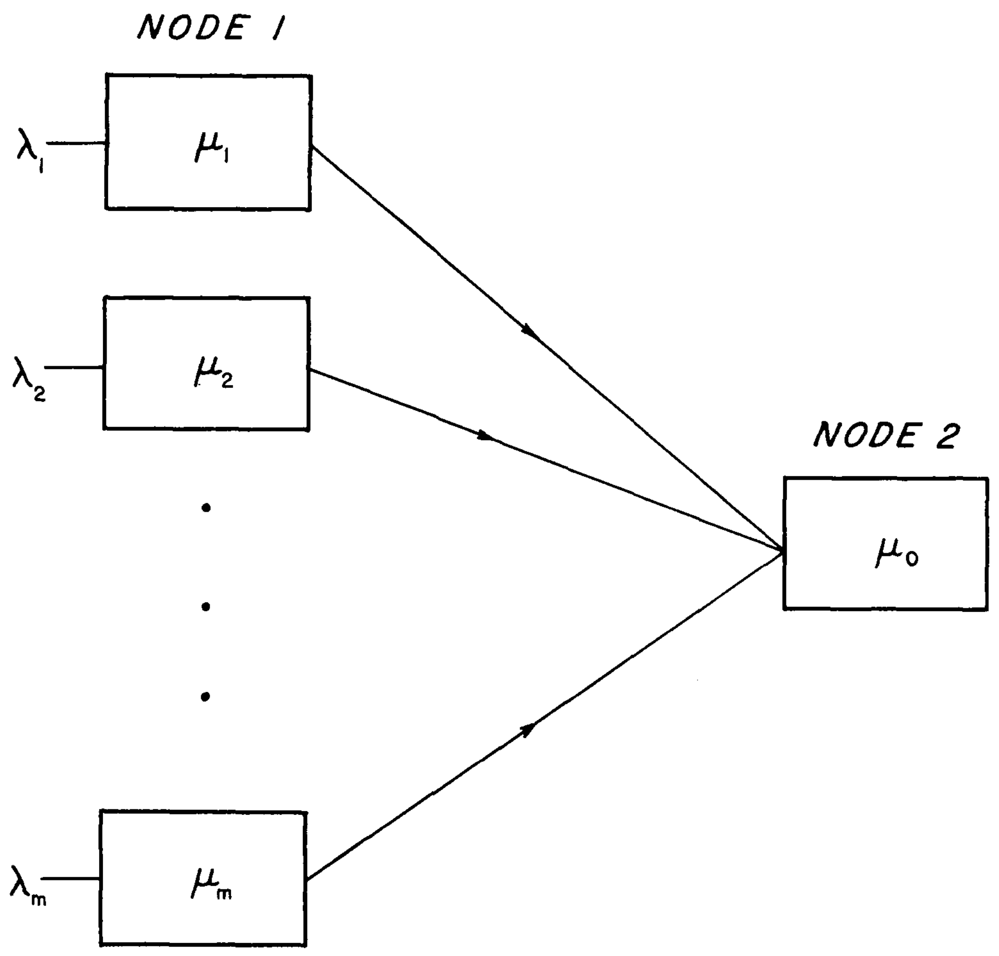
$$(m*Tav)/(k*m*n) < 1$$

$$\text{i.e., } (Tav)/(k*n) < 1$$

From Figure 20b, we have, for stability

$$(Tav)/(k*n) < 1$$

Thus, we derive the same condition for stability from both parts of the model, lending strength to its validity.

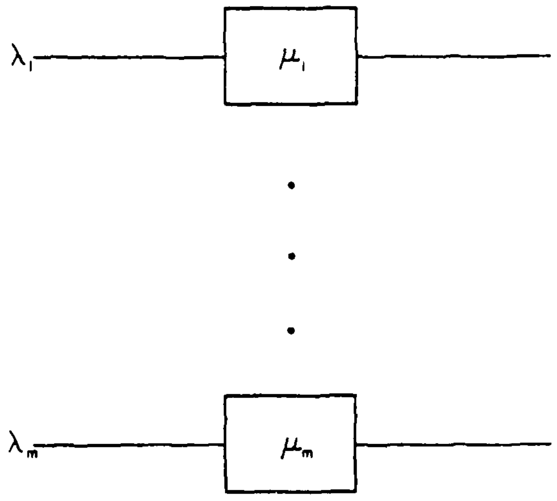


$$\lambda_1 = \lambda_2 = \cdots = \lambda_m = \frac{1}{kmn}$$

$$\mu_1 = \mu_2 = \cdots = \mu_m = \frac{1}{m * L * [P_1 t_1 + P_2 t_2 + P_3 t_3 + P_4 t_4]}$$

$$\mu_0 = \frac{1}{L * [P_1 t_1 + P_2 t_2 + P_3 t_3 + P_4 t_4]}$$

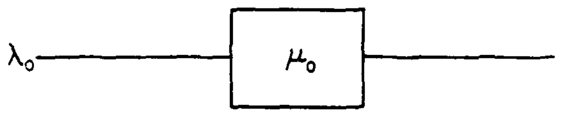
Figure 19. The Two-Node Network of Exponential Servers



$$\lambda_1 = \lambda_2 = \cdots = \lambda_m = \frac{1}{kmn}$$

$$\mu_1 = \mu_2 = \cdots = \mu_m = \frac{1}{m*L*[P_1 t_1 + P_2 t_2 + P_3 t_3 + P_4 t_4]} = \frac{1}{m*Tav}$$

Figure 20a. The First Part of Figure 19



$$\lambda_0 = \frac{m}{kmn} = \frac{1}{kn}$$

$$\mu_0 = \frac{1}{L*[P_1 t_1 + P_2 t_2 + P_3 t_3 + P_4 t_4]} = \frac{1}{Tav}$$

Figure 20b. The Second Part of Figure 19

A. Calculations for Data Loop Without Insert-in-Parallel

We make these calculations based on the following values for the various parameters.

```
p1=p2=p3=p4=0.25
t1=t3=t4=20 milliseconds
t2=40 milliseconds
q=300
p=10
m=10
k=10
```

So, p1*t1 + p2*t2 + p3*t3 + p4*t4 = 25 milliseconds.

Using L = (k*m*n*p*q)/(m*n*p*q - (mp-1)*[p1*t1 + p2*t2 + p3*t3 + p4*t4]) we get, L = (3*(10**5)*n)/(3*(10**4)*n-2475). So, Tav = (75*(10**5)*n)/(3*(10**4)*n-2475). We need, Tav(k*n) < 1; i.e., (75*(10**4))/(3*(10**4)*n-2475) < 1. This tells us that n > 25 milliseconds. We note that the calculations are much simplified by letting k=L. (That is, we will henceforth use T(k)/(k*n) < 1 rather than Tav/(k*n) < 1.) Then, (10*25)/(10*n) < 1. Therefore, n > 25 milliseconds.

This tells us that, without the proposed design, the data loop cannot accept requests at a rate greater than one every 25 milliseconds. Note that changing the value of k causes no improvement in data loop performance. For example, with k=20, we have, (20*25)/(20*n) < 1. This again gives us n > 25 milliseconds.

In the above calculations, we ignored the fact that one extra revolution is needed per set of orders in order to rewrite the bit map. Using this fact, let us redo our calculations.

T(k)=k*[p1*t1 + p2*t2 + p3*t3 + p4*t4] + 20=k*25+20

Therefore, T(k)/(k*n) < 1 implies n > 25 + 20/k. Although the throughput rate achievable is dependent on k, the best achievable throughput is still one that supports an inter-arrival rate of one request every 25 milliseconds.

B. Calculations for Data Loop with the Insert-in-Parallel

Once again, let

```
p1=p2=p3=p4=0.25
t1=t3=20 milliseconds
t2=40 milliseconds
t4=20/(number of insert record requests per set of orders) milli-
seconds. That is, it takes 20 milliseconds to execute all 'insert-
record' orders.
q=300
p=10
m=10
k=10
```

Once again, to simplify the calculations, let k=L. Then, T(k) = L*(p1*t1 + p2*t2 + p3*t3) + 20. That is, T(k) = 220 milliseconds. Applying the condition that T(k)/(k*n) < 1, we get n > 22 milliseconds. Hence, requests may be allowed from the structure memory at a rate of up to one every 22 milliseocnds rather than one every 25 milliseconds (as was the case without the proposed design).

Consider what happens when we make k=L=20. Now we arrive at the condition n > 21 milliseconds. That is, a request arrival rate of one every 21 milliseconds can now be handled. Let us now consider the limiting case when k is made very large. We have, T(k)/(k*n) < 1. That is (k*20 + 20)/(k*n) < 1. Since k*20 >> 20, this is equal to (k*20)/(k*n) < 1. This gives us n > 20 milliseconds. Hence, in the limiting case, the performance of the data loop may be made good enough to handle inter-arrival rates as low as one every 20 milliseconds.

Finally, in Figures 21a and 21b, we show the results of our study in graphical form. These graphs plot the inter-arrival rates that can be handled by the data loop against k (the minimum number of orders that must exist on a cylinder before a seek is initiated on it. This is a design decision.). Figure 21a plots the results of the calculations of the previous paragraphs (i.e., for a distribution where each of the four kinds of requests -- insert, delete, update and retrieve -- has a 25% chance of occurring in a set of orders). The maximum achievable advantage that accrues as a result of the suggested design is 20%. Figure 21b considers a distribution where 50% of the requests are of the insert kind. The maximum achievable improvement in this case is 42.86%.

3.7 The Choice of Record Buffering Size for the TIPs

We see, therefore, that the incorporation of the insert-in-parallel facility can bring us quite dramatic improvements depending upon the distribution of requests. On the basis of the expected distribution of requests, the DBC designer may draw curves similar to those in Figure 21 and thus choose a value of k which gives reasonably good performance. The expected distribution of requests and the value of k that is chosen, will determine the size of the second buffer that is used to store records for insertion. For example, with a distribution where 50% of the requests are 'insert-record' requests, and with k=50, we expect 25 'insert-record' requests per set of orders. Assuming that there are 25 tracks per cylinder, and an even distribution of the 'insert-record' requests across the tracks of the cylinder, each TIP only needs to have a second buffer large enough to hold one record.

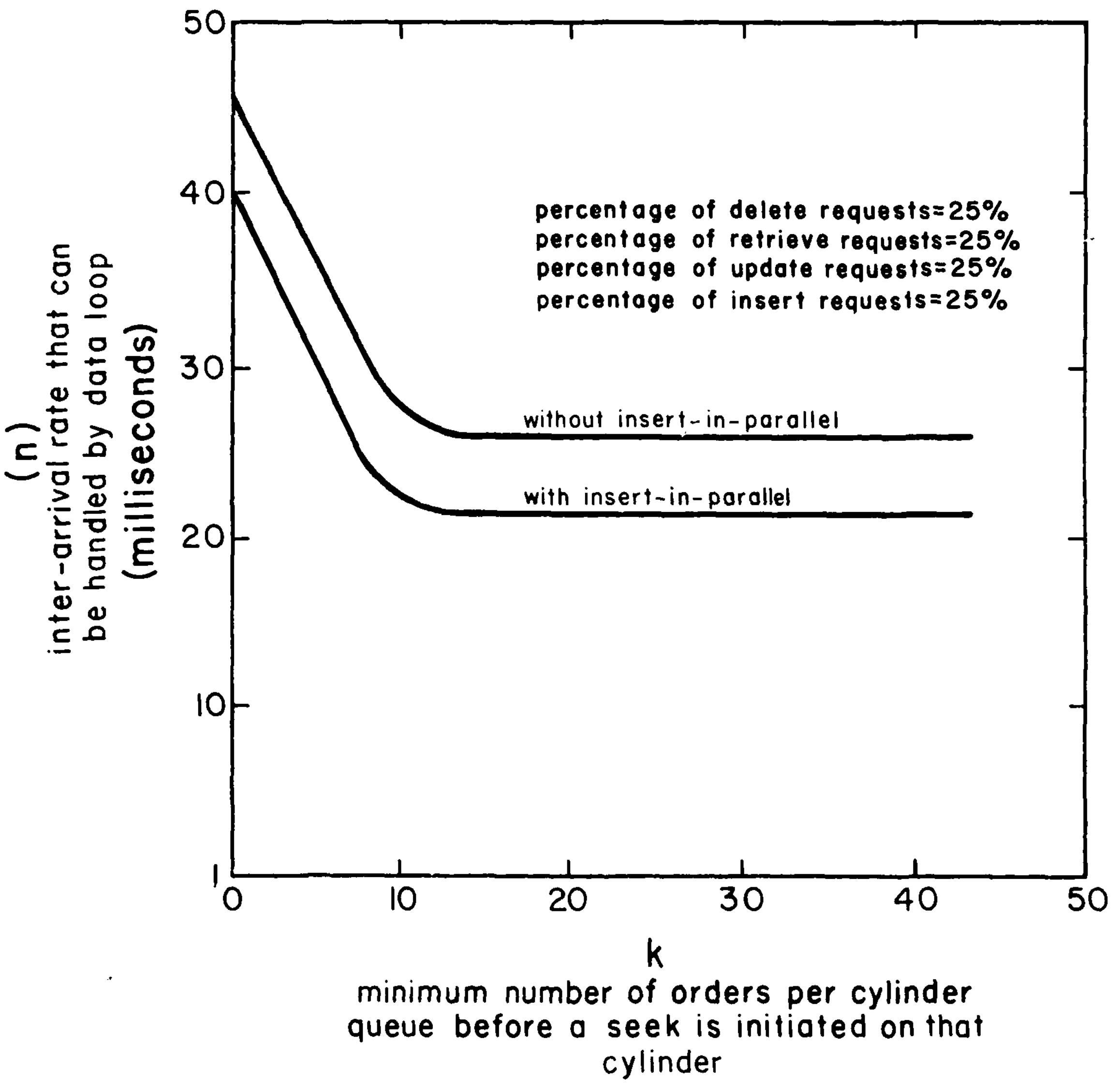


Figure 21a. Graph Showing Results of Study When Percentage of Insert Requests is 25%

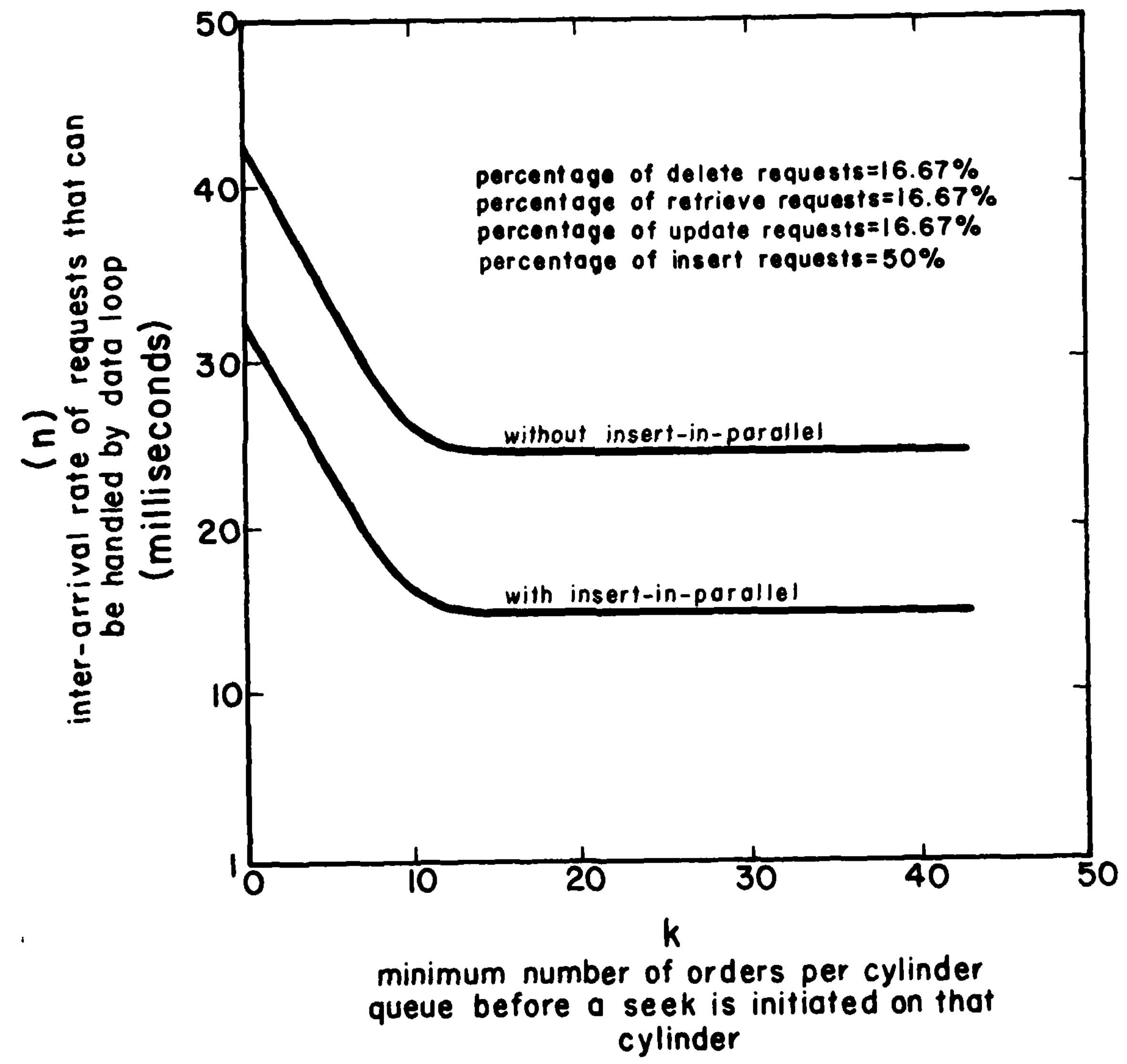


Figure 21b. Graph Showing Results of Study When Percentage of Insert Requests is 50%

4. DELETION OF RECORDS

In this section, we shall examine the method used in DBC to delete records that are stored in the mass memory of DBC. We shall begin by describing certain hardware methods that have been suggested in some other database machines [11], and by explaining why we do not wish to use these methods in DBC. Later, we shall examine the deletion method employed in DBC and propose an improvement of the method. Finally, we shall use the model developed in the previous section to substantiate our claims of performance improvement of the proposed method.

4.1 *The Method Used in CASSM*

We describe here, the method [11,18] used in CASSM (context-addressed segmented sequential memory) for doing deletions. A delete command issued by a user of CASSM does not cause any actual deletion of records. However, records that qualify for deletion are maked as having been deleted, and these marked records are ignored by subsequent commands. The task of repacking these variable-length holes left in memory as a result of deletions is done later by garbage collection hardware.

A register RVL [18] is the basic hardware used in garbage collection. Also, separate read and write heads are needed for each track. As the read head picks up data, it is fed into one end of a shift register RVL which shifts at the same bit rate as the track readout rate. A tap is provided for the output of RVL at multiples of W from the input, where W is the basic word size of the machine. When storage is not being collected (*Figure 22a*), the write head uses the centre tap of RVL as its input. When garbage is being collected (Figure 22b), the input of the write head moves one tap toward the input of RVL each time a word marked for garbage collection is encountered on a track. This eliminates that word from the sequence in memory. It may be easily seen that the number of words that may be garbage collected per revolution is equal to the number of words that can fit into one half of register RVL. If RVL is not long enough to collect all words marked for deletion on a track, they will be collected in subsequent revolutions.

We shall now make a few observations about this method in order to emphasize, at a later time, some of the differences between it and the method that is employed in DBC. First, the above method implies the use of a read head and a write head per track. Second, we note that there has to be at least one mark bit per word. If a record spanning n words is to be marked for deletion, each of the n words of the record must be marked for deletion.

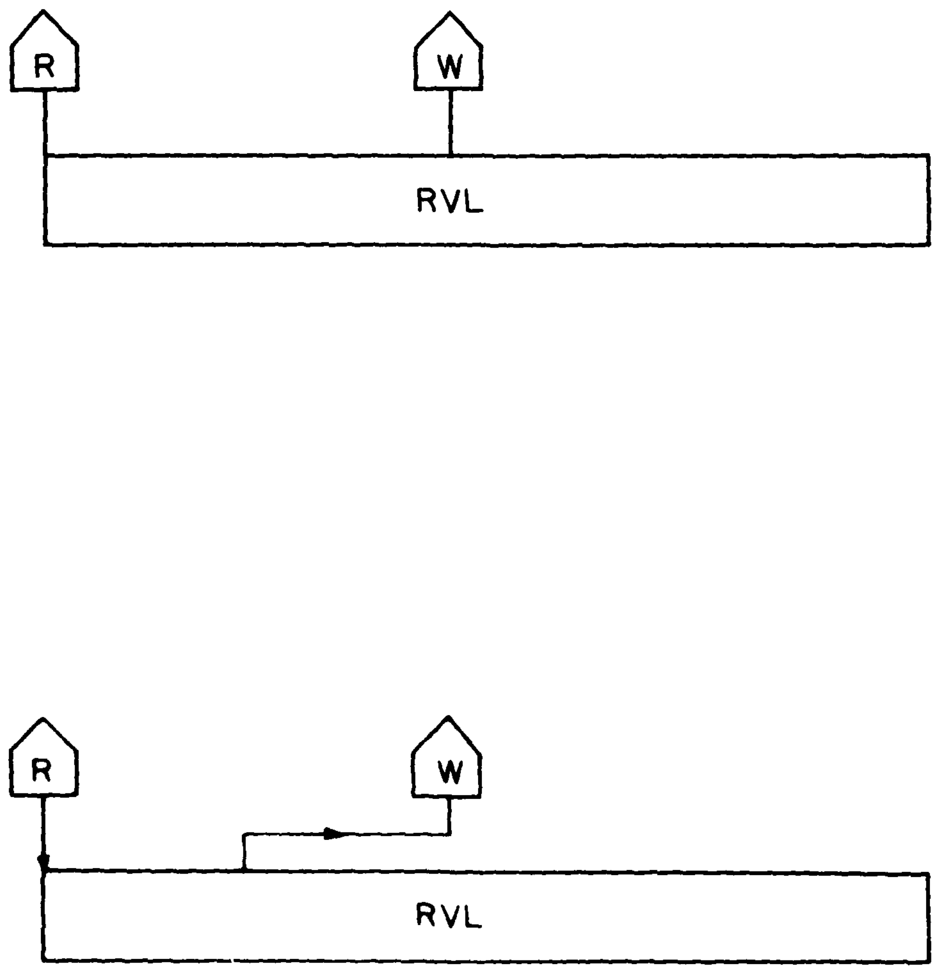


Figure 22. Variable Length Shift Register Used for Deletion of Records in CASSM

is, one mark bit per word rather than one mark bit per record is necessary (in fact, CASSM uses three mark bits per word).

4.2 The DBC Method

We have already stated, in Section 2, why we do not favor using two heads per track. This means, of course, that CASSM's method of garbage collection cannot be used in DBC. The method that is used in DBC to delete records is now described.

We recall, from Section 2, that each track has a bit map stored in the first sector of the track. Each record on a track is represented by a bit in this bit map. The bit is set to '1' if the corresponding record has to be deleted. Before processing of a cylinder is to begin, the bit map in each of the tracks is read by the corresponding track information processor (TIP) into the RAM portion of its buffer. A delete command is accompanied by a query that specifies the records that are to be deleted. Records that satisfy the query are marked as having been deleted by setting the bit corresponding to this record. This may be done 'on-the-fly', since the bit map is stored in a RAM which may be quickly accessed. The following is the algorithm executed by the TIPs in order to process a 'delete-by-query' order.

ALGORITHM: This is executed by the TIPs in order to process a 'delete-by-query' order.

INPUT: The query that specifies the records that are to be deleted.
TIME TAKEN: As many revolutions as there are conjunctions in the query.

```
Step 1:  Let N be the number of conjunctions in the given query. I=1.
Step 2:  Store the I-th conjunction in the query in the query memory of the
         TIPs.  RECORDNUM=1.
Step 3:  If end of track is encountered, then go to Step 4.  Else, start
         reading the bits of record number RECORDNUM from the track and com-
         paring it with the query conjunction stored in the query memory of
         the TIP.  If the record satisfies the conjunction and, hence, quali-
         fies for deletion, set bit RECORDNUM in bit map to '1'.  Go to
         Step 5.
Step 4:  I=I+1.  If I > N, then terminate.  Else, go to Step 2.
Step 5:  When end of record is encountered on track, set RECORDNUM = RECORDNUM + 1
         and go to Step 3.
```

Later retrieve, delete and update commands will ignore records that have their corresponding bit in the bit map set.

Up to this point, there is no essential difference between the CASSM method and the DBC method. Both methods mark records that are meant for deletion. In DBC, only one bit needs to be marked per record to be deleted, and this bit is stored in a RAM and is not part of the record. In CASSM, many bits need to be marked per record to be deleted, and these bits are part of the record itself and are not stored separately.

The difference in the two methods comes about in the compaction (garbage collection) step. DBC enters the compaction mode during periods of light load - when there is a low utilization of system resources. In the compaction mode, MAUs in which tagged (marked) records exist (recall that the mass memory deletion table (MMDT) keeps a list of MAUs that have records tagged for deletion) are accessed, and data in each of the tracks is read into the mass memory controller (MMC) by the TIPs. The MMC then writes back those records which are not tagged. An algorithmic description of the process of compacting records in a MAU is given below.

ALGORITHM: To compact a MAU.

INPUT: The MAU to be compacted.

Step 1: Access the MAU M to be compacted.

Step 2: Request the TIPs to read all tagged records.

Step 3: As the TIPs transmit tagged records over the IOBUS, create a structure memory deletion table (SMDT), a veiw of which is shown in Figure 23. Each entry in this table has two fields. The first field contains a directory keyword, and the second field contains a set of pairs, where each pair is of the form (cylinder number, security atom number). Since this table is temporary and may be deleted at the end of the compaction mode, it may be created in the database object descriptor table (DODT). The SMDT is formed as follows. For each tagged record that is received by the MMC, do the following. For each directory keyword DKi in the record, look to see if there is an entry in the SMDT with DKi in Field 1. If no such entry exists, then create a new entry with DKi in Field 1, and the pair (M,S) in Field 2. [M is the cylinder being compacted and S is the security atom number of the record being examined (the security atom to which a record belongs is part of the information present in a record)]. If such an entry exists, then search the corresponding set of pairs in Field 2 of this entry to see if a pair of the form (M,S) exists. If such a pair exists, then do nothing. Else, add the pair (M,S) to the set of pairs in Field 2 of the corresponding entry. Now discard the record. [In the SMDT, we are putting those SM entries which point to records marked for deletion. Since the records have been deleted, these entries (which point to the deleted records) must also be deleted from the SM].

Step 4: Request the TIPs to read the untagged records. [Since the memory available to the mass memory monitor (MMM) is smaller than the MAU capacity, the MMM will divide the TIPs into sections which are processed sequentially. Thus, if say 80K bytes are available to the MMM and the MAU capacity is 320K bytes, then the TIPs are divided into four sections. TIPs in the same section are requested to read their tracks concurrently during the compaction process. Steps 4 through 7 are repeated for each section].

Step 5: As the records from the TIPs come in, store them in the record storage.

Step 6: For each record in the record storage, do the following. For each directory keyword DKi in the record, look to see if there is an

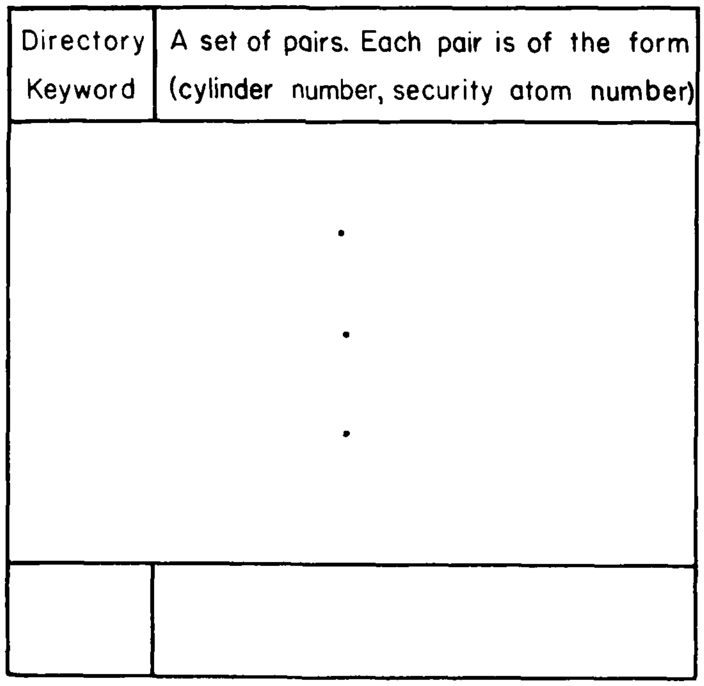

Directory Keyword
A set of pairs. Each pair is of the form (cylinder number, security atom number)

Figure 23. The Structure Memory Deletion Table (SMDT)

entry in the SMDT with DKi in Field 1. If no such entry exists, then do nothing. If such an entry exists, then search the corresponding set of pairs in Field 2 of this entry to see if a pair of the form (M,S) exists. M is the cylinder being compacted, and S is the security atom number of the record being examined. If such a pair exists, delete the pair from Field 2 of the SMDT. Else, do nothing. [In this step, we look at those entries in the SM which point to the untagged (undeleted) records. We wish to delete, from the SM, all entries that point to deleted records. However, some entries in the SM may point to both deleted and undeleted records. We wish to retain these entries in the SM, since, otherwise, pointers to undeleted records will be lost. Hence, any entry in the SMDT which points to an undeleted record is removed from it.]

Step 7: Write the records in the record storage back into the tracks via the TIPs.

Step 8: Pass the SMDT table to the database command and control processor (DBCCP). DBCCP then accesses the structure memory (SM) and deletes the relevant index terms from it.

We will now estimate the time taken to compact a MAU. First, all the tagged records are read and processed. Since the reading of one record (by the TIPs) occurs at the same time as the processing of another record (by the MMC), one revolution of the disk device will suffice for this. Next, the untagged records are read, processed and written back, a section at a time. Once again, the time to read and process a section is one revolution. However, the time taken to write back a section is very large (without insert-in-parallel) since writing back has to be done a record at a time. Assuming 80K bytes per section and 1K byte records, it takes 80 revolutions to write back a section! If the TIPs are divided into r sections, the time taken to compact a MAU is

$$1 + r*(1+80) = 1+81r \text{ revolutions.}$$

Typical values of r range from 3 to 6, so that the time of compaction ranges from 244 revolutions to 487 revolutions.

However, with insert-in-parallel, only one revolution is needed to write back a section. Thus, the time taken to compact a MAU is

$$1 + r*(1+1) = 1 + 2r \text{ revolutions.}$$

Once again, letting r range from 3 to 6, we see that the time of compaction ranges from 7 to 13 revolutions. This is an improvement of about 97%. We thus see that allowing for insert-in-parallel tremendously improves the performance of DBC during the compaction mode of operations.

4.3 Further Improvements

Before we examine if it will be possible to further improve the time taken to compact a MAU, let us examine the reasons for choosing to implement the process of compaction in the manner described in the last sub-section. If we were to provide a track-size buffer with each TIP, then compaction may be done in the TIPs itself without having to move records to and from the MMC. This will enable us to do compaction more quickly than possible now. However, our original reasons for not having track-size buffers with each TIP was because these buffers would be underutilized during the retrieval operation. We would now like to reexamine our motives for not choosing to have track-size buffers for each TIP.

First of all, with memory technology making rapid advances, under-utilization of memory is no longer as important as it used to be. Secondly, we are now trying to design DBC to operate in an update-intensive environment where there are many requests for deletion. Therefore, we would like to enhance DBC performance during the compaction phase even if it means underutilization of memory. That is, we are willing to pay the price of underutilization in an update-intensive environment but not in an update-free environment. Finally, in an update-intensive environment, the underutilization of TIP buffers will not be a problem, since, the precentage of retrieval requests is expected to be very low. The remaining portion of this section will be devoted to showing how the performance of DBC may be improved, during the compaction phase, by using track-size buffers with each TIP. In the next section, we will show that track-size buffers will be needed to improve the performance of DBC during execution of update commands. Thus, we see that the buffers will be under-utilized only during execution of retrieval commands, and in an update-intensive environment, there are expected to be very few retrievals. For all the above reasons, we advocate the use of track-size buffers with each TIP.

A new method of compaction is now proposed for DBC that utilizes the track-size buffer. A description of this method follows. During the compaction mode, the TIPs read records in their respective tracks and place all untagged records in one buffer (say Buffer A) and all tagged records in another buffer (say Buffer B). [Earlier, we had indicated the need for two different buffers. Buffer A is the buffer that is used during the execution of retrieve and update commands and is the size of a track as we have postulated above. Buffer B is the buffer that is used to hold records for insertion. In Section

3, we had shown how to estimate the size of this buffer depending on various factors such as average cylinder queue size and the percentage of insert requests.] This reading of both tagged and untagged records may be done in one revolution of the disk device. Recall, that a TIP consists of a disk interface processor (DIP) and a control interface processor (CIP). In the first revolution, as the DIP is reading the tagged and untagged records, the CIP is processing the tagged records in Buffer B. In the second revolution, the DIP writes back all the untagged records in Buffer A, even as the CIP processes the untagged records in Buffer A. Since each buffer consists of individually accessible segments, the CIP can process those segments of Buffer A which have been written back by the DIP. We will now describe the nature of the processing done by the CIPs, during the two revolutions, in an algorithmic fashion. It will be noted that each TIP needs an additional RAM to do this processing. This RAM will store two tables, the tagged deletion table (TDT) and the untagged deletion table (UDT). Both these tables are similar to the structure memory deletion table (SMDT) which was described in the previous sub-section (see Figure 23).

```
ALGORITHM:  Executed by the CIP during the first revolution of the compaction
phase.

Step 1:     I = 0.
Step 2:     I = I + 1.  Read the I-th tagged record in Buffer B.  If there is no
            such record in Buffer B, then terminate.
Step 3:     For each directory keyword DKi in the record, look to see if there is
            an entry in the TDT with DKi in Field 1.  If no such entry exists,
            then create a new entry with DKi in Field 1, and the pair (M,S)
            in Field 2.  [M is the cylinder being compacted, and S is the security
            atom number of the record being examined.]  If such an entry exists,
            then search the corresponding set of pairs in Field 2 of this entry
            to see if a pair of the form (M,S) exists.  If such a pair exists,
            then do nothing.  Else, add the pair (M,S) to the set of pairs in
            Field 2 of the corresponding entry.  Go to Step 2.
```

In the first revolution, the DIP reads all tagged records into Buffer B and all untagged records into Buffer A. In the same revolution, the CIP builds up, in the TDT, a list of SM entries which point to deleted records. Naturally, we shall remove these entries from the SM as long as they do not also point to undeleted records. In revolution two, the CIP executes the same algorithm as above except that it processes the records in Buffer A (rather than those in Buffer B), and that it utilizes the untagged deletion table (UDT) and not the tagged deletion table (TDT). That is, the CIP builds up, in the UDT, a list of SM entries which point to undeleted records. This same revolution is utilized by the DIP to write back all undeleted records.

Thus, at the end of two revolutions, all the untagged records have been written back on to the track and each TIP has created two tables, the TDT and the UDT. These tables are now sent to the MMC, which then processes these tables, in the third revolution, to form the structure memory deletion table (SMDT) as follows.

ALGORITHM: Executed by the MMC to form the structure memory deletion table (SMDT).

INPUTS: One TDT and one UDT per TIP.

Step 1: FLAG=0. Concatenate all the TDTs into a single TDT. Let there be N entries in the newly created TDT. Set I=1.

Step 2: If FLAG=0, then examine the I-th entry in the TDT. Else, examine the I-th entry in the UDT. If the I-th entry is null, then go to Step 4. If the I-th entry is nonnull, let it be of the form <value, S>, where S is a set of pairs. Look through entries I+1 to N, looking for entries which have the same value in Field 1 as the I-th entry has. Put these entries into a set Q. For each entry in set Q, do Step 3.

Step 3: Let the entry be of the form <value, R>, where R is a set of pairs. Delete this entry (i.e., make it a null entry) from the TDT (delete this entry from the UDT if FLAG = 1) and change the I-th entry of the TDT (change the I-th entry of the UDT if FLAG = 1) to <value, $S \cup R$>.

Step 4: I=I+1. If $I > N$ & FLAG=0, then go to Step 5. If $I > M$ & FLAG=1, then go to Step 6. Else, go to Step 2.

Step 5: FLAG=1. Concatenate all the UDTs in to a single UDT. Let there be M entries in the UDT. Set I=1 and go to Step 2.

Step 6: Remove all null entries from the TDT and the UDT. [At this point, we have merged all the entries in the TDTs into a single TDT and all the entries in the UDTs into a single UDT.]

Step 7: For each entry in the UDT, do the following. Let the UDT entry be <value, X>. Search the TDT looking for an entry with the same value in Field 1 as the entry in the UDT. Let this be the entry <value, Y>. Recall that X and Y are sets of pairs Delete the UDT entry, and change the TDT entry to <value, Y-X>. [Recall, that we do not want to delete all the entries present in the TDT from the SM. This is because some of these entries may also point to undeleted records. In this step, we remove all entries in the TDT which are also present in the UDT. Since entries in the TDT point to deleted records and entries in the UDT point to undeleted records, it follows that entries common to both the TDT and the UDT point to both deleted and undeleted records. Hence, we are keeping, in the TDT, only those entries which point solely to deleted records. These entries must be removed from the SM.]

Step 8: The resulting TDT is called the structure memory deletion table (SMDT). This SMDT is now sent to DBCCP so that the corresponding index terms may be deleted from the structure memory.

Note that, in the above method of compaction, the TDTs and the UDTs do not have to be sent to the MMC a section at a time. This is because the size of these tables is much smaller than the size of all the untagged records in a track. Thus, the entire compaction process now takes approximately three revolutions. We have thus reduced the process of compaction from a maximum of 487 revolutions, to a minimum of 3 revolutions.

4.4 Elimination of Compaction Mode and Deletion Bit Maps

The careful reader would of course have realized that garbage collection would be unnecessary if records were of fixed length. This is because insertions may then be done in slots left vacant by deletions, since all records, being of fixed length, would fit into these empty slots. However, when records are of variable length, one record may not fit into the slot left vacant by the deletion of another record and, hence, compaction is necessary. Actually, it is not necessary that all records be of fixed length in order to avoid the overhead of garbage collection. It is only necessary that all records in a file be of fixed length. This is because a cylinder only contains records from a single file. Since most of the files encountered in the real world have fixed-length records, no compaction will be necessary in most of the cases. In DBC, we will compact only those MAUs which contain variable-length records.

Notice, that changes to the index terms in structure memory (SM) can be made only after the process of compaction has created the SMDT. Let us see how this will affect the execution of requests that are issued after a delete command, but before the necessary changes (caused by the delete request) have been made to the index terms in SM. Basically, the problem is the existence in SM of index terms that ought not to be there. This could cause some deleted index terms to be retrieved from SM in response to a query. Since index terms consist of (cylinder number, security atom) pairs, this may lead to extra accesses to cylinders (i.e., cylinders in which no record that can satisfy the given query exist). The impact of this can, of course, be reduced by doing compaction more often.

A logical time to do compaction on a cylinder would be at the end of execution of a set of orders on that cylinder. We would like to get some quantitative figures on the drop in throughput rate of the data loop as a result of doing compaction at the end of a set of orders. We shall use the model developed in Section 3 to come up with some quantitative results.

4.4.1 Calculations of Data Loop Throughput

Let,

$p1=p2=p3=p4=0.25$ (i.e., an even distribution of requests)
$t1=t3=20$ milliseconds.
$t2=40$ milliseconds.
$q=300$
$p=10$
$m=10$
$k=L=10$

If we do not compact at the end of a set of orders,

T(k) = k*(p1*t1 + p2*t2 + p3*t3) + 20 milliseonds.
= 220 milliseconds

If we do compact at the end of a set of orders,

T(k) = k*(p1*t1 + p2*t2 + p3*t3) + 80 milliseconds.
= 280 milliseocnds

This is becuase three additional revolutions are needed for compaction. Applying the condition T(k)/(k*n) < 1, we get n > 22 milliseconds without compaction at the end of every set of orders, and n > 28 milliseconds with compaction at the end of every set of orders. However, as k gets very large, the limiting condition, in both cases, is n > 20 milliseconds. That is, we may compensate for the extra time spent in compacting at the end of a set of orders by increasing k (the minimum number of requests that must be present on a cylinder queue before a seek may be issued on that cylinder). Figure 24 is a graph of k versus the maximum inter-arrival rate that the data loop can sustain. We note that effects of compaction on the throughput rate are minimal when k is made very large. Thus, by making k fairly large and compacting at the end of every set of orders, we can minimize the number of unnecessary cylinders that might have to be searched owing to the presence of an index term that ought to have been deleted.

Let us examine some of the implications of doing compaction at the end of every set of orders. First, this means that DBC no longer needs to switch to the compaction mode in order to do compaction. That is, compaction is done, in the normal mode itself, by incurring three extra revolutions at the end of a set of orders. This, as the graphs show, will not affect the throughput rate on the data loop if a large enough value of k (the minimum number of orders that must exist on a cylinder before a seek is initiated on it) is chosen. Second, we have shown that the process of compaction builds up a SMDT which contains all entries in the SM which must be deleted. Postponing the process of compaction until the DBC encounters periods of light load would cause a poor response to many requests. This is because the presence of indices in the SM which ought to have been deleted would cause the access of cylinders which do not contains records that satisfy the query in the request. Thus, many unnecessary seeks would be carried out in DBC This is avoided by doing the compaction as often as possible. Finally, we note that it is no longer necessary to store the bit map on the track itself. The bit map only needs to be in the RAM associated with each TIP. When compaction is not done at the end of every

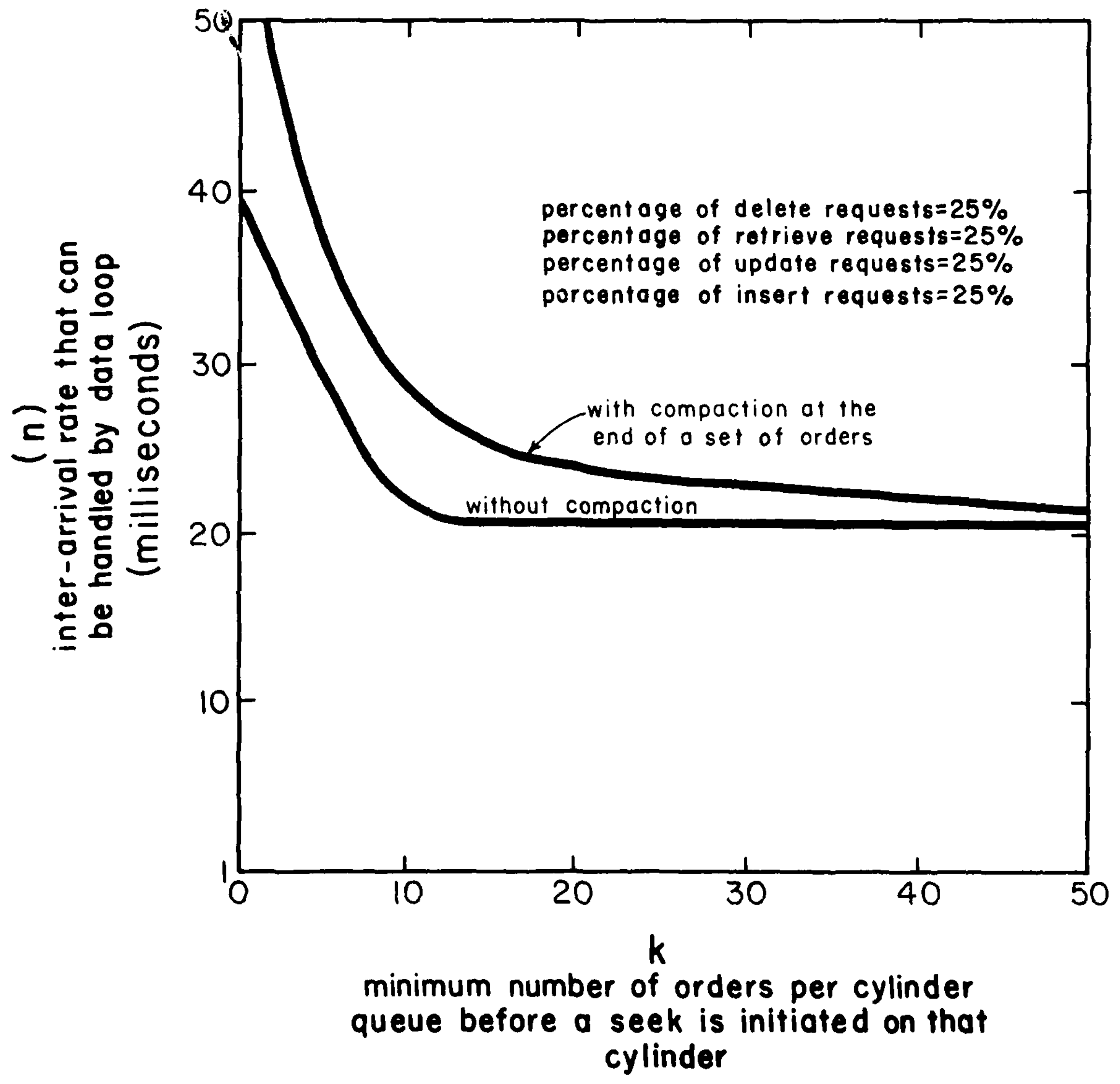


Figure 24a. Graph Showing Results of Study When Percentage of Insert Requests is 25%

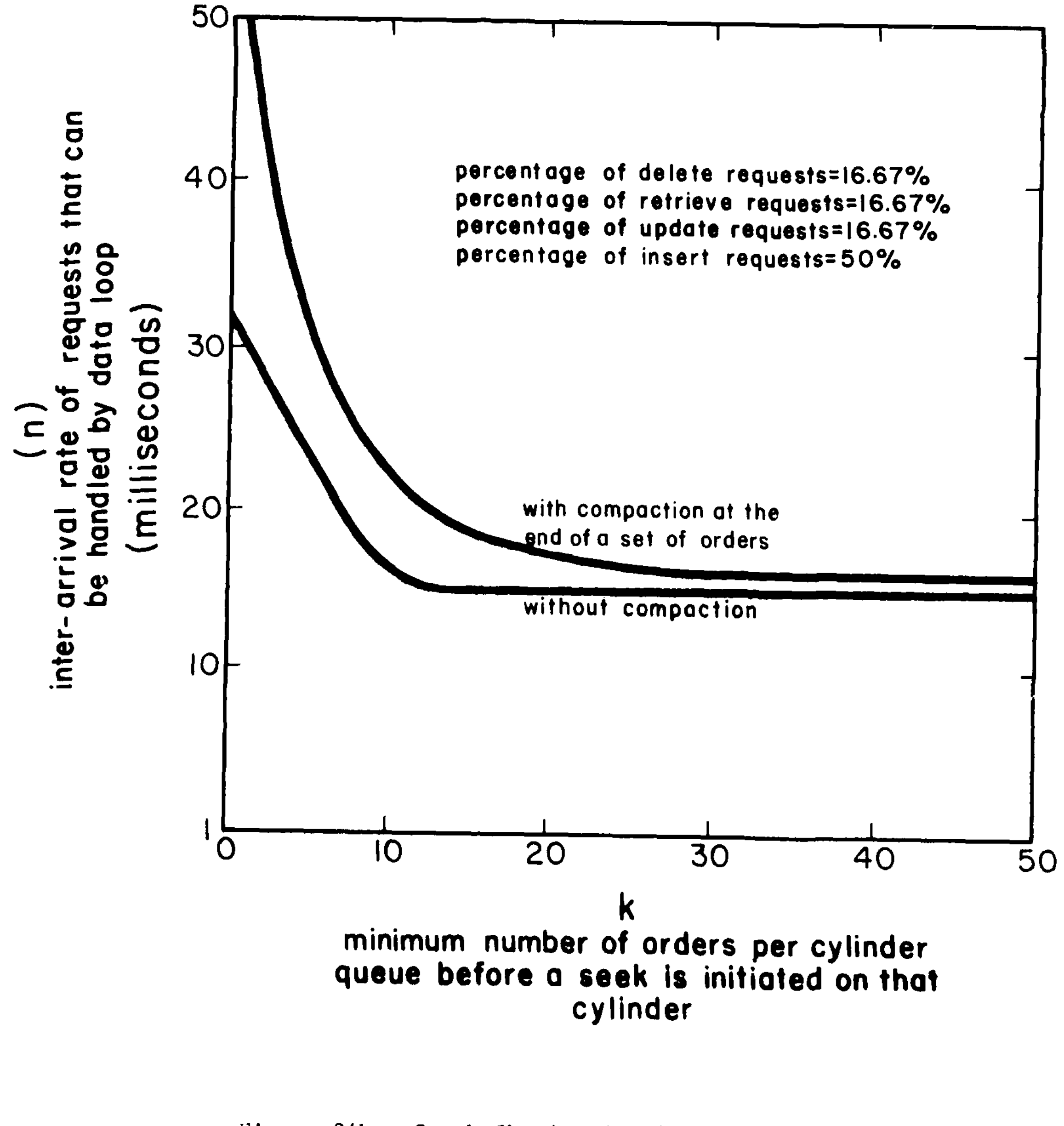


Figure 24b. Graph Showing Results of Study When Percentage of Insert Requests is 50%

set of orders, it is necessary to copy the bit map from the RAM onto the track in order to remember the positions of deleted records. However, when compaction is done at the end of a set of orders, all the deleted records have been removed and, hence, it is unnecessary to remember the positions of deleted records. This implies that more space is available to store records in each track.

5. UPDATING OF RECORDS

In this section, we shall take a look at the process of updating records stored in DBC. We shall begin by describing the exact nature of the update requests. That is, the kinds of update that are allowed in DBC are clearly specified. These allowed updates are then classified into various categories depending upon whether the part of the record that is updated is a simple keyword, a directory keyword, a clustering keyword or a security keyword. This is because these different categories of update need different amounts of processing by DBC. The processing of update requests is then described assuming that the size of Buffer A (see Section 4) is less than track-size the performance of the data loop without track-size buffers is compared to its performance when track-size buffers are used. Finally, a proposal is made for the purpose of improving the data loop performance to a greater degree.

5.1 The Nature of Update Commands

In DBC, the format of an update command is as shown below.

UPDATE   QUERY   MODIFIER

That is, the command consists of three parts. The first part specifies the command name. The second part is a query. All records that satisfy this query must be updated. Optionally, the second part may contain 'ALL' to indicate that all records in DBC are to be updated. The modifier part specifies the kind of modification that needs to be done on the records that satisfy the query. We have already indicated, in Section 2, the exact nature of a query which will not be reproduced here. Instead, we shall concentrate on the nature of the modifier. The modifier is essentially of the form

<attribute, newval = f(oldval)>

That is, it is a pair. The first part of the pair specifies an attribute which is present in the keywords of the records that are to be modified. This attribute is called _the attribute to be modified_ and the pair <attribute, oldval> is the _keyword to be modified_. The second part of the pair specifies the value that this attribute must take (newval) in the updated records as a function of the value that it had before the update (oldval). We note that both newval and oldval are literals, or reserved words of DBC. Examples of types of modifiers allowed in DBC are indicated below.

1. <Name, newval = HSIAO>.

Here, the new value that is to be taken by the Name attribute is 'HSIAO'. This is the constant function, since it is not a function that depends on the old value of the Name attribute.

2. <Salary, newval = oldval+5000>.

Here, the salary of employees (as identified by the query part) will be increased by $5000.

3. <Salary, newval = oldval*110/100>.

Here, all employees (identified by the query part) are given a 10% raise.

In DBC, we may also allow the modifier part of the update command to be of the following form.

<attribute1, newval1=f(<attribute2, oldval2>)>.

That is, the new value (newval) that an attribute (attribute1) is to take is a function of the old value (oldval2) of another attribute (attribute2) in the records that are to be updated (as identified by the query part). The <u>attribute to be modified</u>, in this case, is attribute1. The pair <attribute1, oldval1> is the <u>keyword to be modified</u>. We shall refer to attribute 2 as the base attribute of modification. Once again, newval1 and oldval2 are literals. An example of a modifier that fits the above format is

<Monthsal, newval1 = <Yearsal, oldval2>/12>.

This modifier causes the monthly salary earned by employees in records identified by the query part to be a twelfth of the yearly salary earned by these employees. The attribute to be modified is Monthsal and the base attribute of modification is Yearsal.

Hencforth, we shall say that the modifier in an update command can be of one of the following three types.

TYPE 0: <attribute, newval=constant>
TYPE I: <attribute, newval-f(oldval)>
TYPE II: <attribute1, newval1=f(<attribute2, oldval2>)>

We note that DBC update command allows only for the update of a single attribute's value. If many attribute values have to be updated, many DBC update commands will be needed.

5.2 The Concept of Blocking

Before classifying updates into various categories, let us explain, by means of an example, some of the problems associated with record updating. Let us imagine a particular database with many records. One of the attributes that appears in some of these records is, say, Salary. A view of the structure memory (SM) content is shown in Figure 25. Each entry in the SM, as we recall, is a keyword or keyword descriptor followed by a set of indices. Each index is a (cylinder number, security atom number) pair. The first SM entry in Figure 25 indicates that records containing the keyword <Salary, 75> are present in Cylinders 2 and 3, and that these records belong to Security Atom number 2. The second entry in Figure 25 indicates that records containing the keyword <Salary, 50> are stored in Cylinders 1 and 4, and that they belong to Security Atom number 1.* Consider, now, that the following two requests are issued to DBC, one after the other.

(1) UPDATE ALL <Salary, newval=oldval+25>

(2) RETRIEVE (Salary=75)

The first command is an update command. It increments the Salary of all records by $25. After execution of this command, all records that originally contained the keyword <Salary, 75> will contain the keyword <Salary, 100>. Similarly, all records that contained the keyword <Salary, 50> will contain the keyword <Salary, 75>. After the records in the mass memory are properly updated, the structure memory must also be updated. This is depicted in Figure 26.

Consider what happens if the second request, i.e., the retrieve request, is executed before SM is finished with the update. In other words, the records in MM are properly updated and the entries in SM have not yet been modified. The second request requires the retrieval of all records that satisfy the query (Salary = 75), i.e., all records that contain the keyword <Salary, 75>. Since the update process is not yet completed, SM may still indicate that such records are present in Cylinders 2 and 3 (see Figure 25 again). The mass memory will now execute this request by accessing each of these cylinders, in turn, looking for records that contain the keyword <Salary, 75>. However, due to the earlier update command executed by DBC, only records that contain the keyword <Salary, 100> are present in Cylinders 2 and 3. Thus, the MM will not be able to retrieve a single record with the keyword <Salary, 75>. We know, of

*See note in Figure 25.

SM Entries *	
<Salary, 75>	(2,2),(3,2)
<Salary, 50>	(1, 1),(4,1)
•	•
•	•
•	•

* Normally, each SM entry may cover a range of values, say, $75 \leq \text{Salary} < 1000$. Very seldom do we utilize SM for storing indices of discrete attribute values. For the simplicity of this illustration, we use discrete attribute values.

Figure 25. A View of the Structure Memory (SM), before Execution of the Update Command

SM Entries	
<Salary,100>	(2,2),(3,2)
<Salary, 75>	(1, 1),(4,1)
•	•
•	•
•	•

Figure 26. A View of the Structure Memory (SM) after it has been Modified to Reflect the Changes Caused by the Update

course, that records which contain the keyword <Salary, 75> are present in Cylinders 1 and 4.

There are two distinct problems owing to the execution of the next command before the completion of the update command.

(1) It has caused the accessing and search of cylinders which do not contain records that satisfy the second request.

(2) It did not access and search cylinders which did contain records satisfying the request.

Both problems lead to a decrease in recall (i.e., the number of retrieved records that satisfy the user query vs. the total number of records in the database that satisfy the query). For DBC, we require total recall. Hence, we must ensure the following. Following an update request that modifies a directory keyword, we must block the execution of all subsequent requests that may be affected by the update request until the structure memory has been modified to reflect the changes in the MM caused by the update request. We shall define thar we mean by requests that may be affected by an update request in the following sub-sections.

5.2.1 Update of Nonsecurity and Nonclustering Attributes

Consider a specific update request in which the attribute to be modified is part of a directory keyword, but is neither a security attribute nor a clustering attribute.

UPDATE   QUERY   MODIFIER

Let the attribute to be modified be attribute1. Any susequent retrieve (or delete) request issued will use a query to specify the records to be retrieved (or deleted). If any of the subsequent requests contains a query that uses attribute1, then this retrieve (or delete) request may be affected by the update. More specifically, a subsequent retrieve (or delete) command will be affected by the update only if the query used in the retrieve (or delete) command contains attribute1 and the set of records that satisfy this query overlaps with the set of records that satisfy QUERY. We are therefore required to determine whether the set of records that satisfy one query will overlap with the set of records that satisfy another query. Unfortunately, it is not always possible to tell, given two queries, whether the records that satisfy one query will overlap with the records that satisfy another query prior to any record retrieval. However, there are ways to tell, given two queries, that the set of records that satisfy one query will not overlap with the set of records that satisfy the other query. For example, it is easy to see that records that satisfy the query (Salary < 50) will not overlap with records that satisfy the query (Salary > 50). However, given the query (Salary = 50) and the query (Name = HSIAO), there is no way to tell, without an actual examination of all the records in the database, whether the set of records that satisfy one query will overlap with the set of records that satisfy the other query, i.e., there is no way of telling if there exists at least one record that satisfies both queries.

We shall refer to two queries such that the set of records that satisfy one query overlaps with the set of records that satisfy the other query as clashing queries. If two queries are clashing, all the attributes in either query are called clashing attributes. On the other hand, (Salary>50) and (Salary<50) are nonclashing queries and Salary is the nonclashing attribute. We observe that two queries are nonclashing if at least one of the attributes in either query is nonclashing. Thus, ((Name=HSIAO) & (Salary>75)) and (Name= HSIAO) & (Salary<75)) are nonclashing queries even though the two queries may be clashing on the attribute Name. If we cannot make certain that two queries are nonclashing, we say that they may be clashing.

Given two queries, we will try to determine in DBC if they are definitely nonclashing on one or more attributes. A means to do this will be elaborated in subsequent paragraphs. If it cannot be definitely determined that two queries are nonclashing on some attribute, we conclude that the two queries may be clashing on all attributes in the two queries.

Consider the following example and let the update be UPDATE1 and subsequent requests be RETRIEVE. We have

UPDATE (attribute1>50) <attribute1, newval1=oldval1-25>
RETRIEVE (attribute1<50)

Since the query (attribute1<50) and the query (attribute1>50) are nonclashing on the attribute attribute1, we might wrongly conclude that the records that satisfy one query will not overlap with the records that satisfy the other query. However, this is not true. This is because during the update the first request has decreased the value of attribute1 in some of the records in the database. Thus, these records which originally had a value for attribute1 greater than 50 might now have a value for attribute1 less than 50. Hence, the second request could refer to some records that were updated by the earlier request and hence may be affected by UPDATE1. For a second request to be unaffected by an earlier update request, it is not enough to see if the queries in the two queries are nonclashing on an attribute other than attribute1 (i.e., the attribute to be modified in the update request). We can conclude that a retrieve (delete or update) request issued subsequent to an update is unaffected by the update only if either the query in the request does not contain the attribute to be modified or the query in the request is nonclashing with the query in the update on an attribute other than the attribute to be modified.

Insertion requests issued subsequent to an update are not affected by the update.

We are now in a position to give a complete and formal definition of what we mean by 'may be affected'. A request issued subsequent to UPDATE1 [UPDATE QUERY1 <ATTIRBUTE1, NEWVAL=CONSTANT>] or [UPDATE QUERY1 <ATTRIBUTE1, NEWVAL1=f(OLDVAL1)>] or [UPDATE QUERY1 <ATTRIBUTE1, NEWVAL1=f(<ATTRIBUTE2, OLDVAL2>)>], where ATTRIBUTE1 is not a security or clustering attribute, may be affected by UPDATE1 if one of the following is true.

(1) It is a retrieve request and the query used in the request contains ATTRIBUTE1 and may be clashing with QUERY1 on all attributes in the two queries other than ATTRIBUTE1.

(2) It is a delete request and the query used in the request contains ATTRIBUTE1 and may be clashing with QUERY1 on all attributes in the two queries other than ATTRIBUTE1.

(3) It is an update request of the form UPDATE QUERY MODIFIER and QUERY2 may be clashing with QUERY1 on all attributes in the two queries other than ATTRIBUTE1 and ATTRIBUTE1 is part of QUERY2.

All requests that may be affected by a prior update command are called blocked requests. The prior update command which causes other requests to be blocked is called the blocking request.

Let us explain the definition of the previous paragraph by means of some examples. Consider the following update request.

1. UPDATE ((Salary>50) ∧ (Name=HSIAO)) <Salary, newval=oldval+50>.

Let us assume that the following set of requests was issued subsequent to the issuance of the update request above.

(a) RETRIEVE (Name=HSIAO)

(b) RETRIEVE (Name≠HSIAO)

(c) DELETE (Salary=75)

(d) RETRIEVE (Salary<25)

(e) DELETE (Salary>50) ∧ (Name≠HSIAO)

(f) UPDATE (Name=HSIAO) <Dept, newval=oldval*2>

(g) UPDATE (Name=HSIAO) <Salary, newval=oldval*2>

(h) UPDATE (Name≠HSIAO) <Salary, newval=oldval*2>

(i) UPDATE (Salary>50) <Salary, newval1=<Rent, oldval2>>

(j) UPDATE (Name=HSIAO) <Rent, newval1=<Salary, oldval2>>

Let us now consider each of the requests (a) through (j) in turn, and see which of them may be affected by the update request shown in (1). We note that, in this update request, ((Salary>50) ∧ (Name=HSIAO)) is the update query and Salary is the attribute to be modified.

(a) This is a retrieve request with the query (Name=HSIAO). This query does not contain the attribute to be modified by update request, *i.e., Salary. Hence,* in spite of the fact that the query (Name=HSIAO) may be clashing with the query ((Name=HSIAO) ∧ (Salary>50)) on the Name attribute, this request is unaffected by the update request.

(b) This request is also unaffected by the update request because query (Name≠HSIAO) does not contain the attribute Salary.

(c) This request is a delete request and it is accompanied by the query (Salary=75) which obviously contains the attribute Salary. Also, the query (Salary=75) may be clashing with the query ((Salary>50) ∧ (Name=HSIAO)) on the Name attribute (since we cannot determine that it is nonclashing on the Name attribute). Hence, this request may be affected by the update request.

(d) This request is a retrieve request and it is accompanied by the query (Salary<25) which obviously contains the attribute Salary. Also, the query (Salary<25) may be clashing with the query ((Salary>50) ∧ (Name=HSIAO)) on the Name attribute (the fact that they are nonclashing on the Salary attribute is unimportant since Salary is the attribute to be modified in the update request). Hence, this request may be affected by the update request.

(e) This is a delete request which is accompanied by the query ((Salary>50) ∧ (Name≠HSIAO)). Even though this query contains the

Salary attribute, it is nonclashing with the query ((Salary>50) $\wedge$ (Name=HSIAO)) on the Name attribute. Hence, this request will be unaffected by the update request.

(f) This is an update request with query (Name=HSIAO) and a modifier of TYPE I, where the attribute to be modified is Dept. We note that (Name=Hsiao) may be clashing with ((Name=HSIAO) $\wedge$ (Salary>50)) on the Name attribute. However, Salary is not part of the query (Name=HSIAO). Therefore, this update request is not affected by the update request.

(g) This is an update request with query (Name=HSIAO) and a modifier of TYPE I, where the attribute to be modified is Salary. The query (Name=HSIAO) may be clashing with ((Name=HSIAO) $\wedge$ (Salary>50)) on the Name attribute. However, Salary is not part of the query (Name=HSIAO). Hence, this update is not affected by the update request.

(h) This is an update request with query (Name≠HSIAO) and a modifier of TYPE I, where the attribute to be modified is Salary. However, since (Name≠HSIAO) is nonclashing with ((Name=HSIAO) $\wedge$ (Salary>50)) on the Name attribute, this request will be unaffected by the update request.

(i) This is an update request with query (Salary>50) and a modifier of TYPE II, where the attribute to be modified is Salary, and the base attribute of modification is Rent. First, the query (Salary>50) may be clashing with the query ((Salary>50) $\wedge$ (Name=HSIAO)) on the Name attribute. Second, the attribute Salary is part of the query (Salary>50). Hence, this request may be affected by the update request.

(j) This is an update request with query (Name=HSIAO) and a modifier of TYPE II, where the attribute to be modified is Rent., and the base attribute of modification is Salary. Even though the query (Name=HSIAO) may be clashing with the query ((Salary>50) $\wedge$ (Name=HSIAO)), Salary is not part of the query (Name=HSIAO). Hence, this request is not affected by the update request.

We conclude, from the examples above, that the execution of requests c, d, and i must be staved until the structure memory is fully updated to reflect the changes caused by the update request. That is, requests c, d, and i are blocked by the update request.

5.2.2 Updating Security and Clustering Attributes

Up to this point, we had considered the requests blocked by an earlier update request, where the attribute to be modified in the earlier update was not a security or a clustering attribute. In order to illustrate how the situation is different if the attribute to be modified is either a security or a clustering attribute, we resort to an example.

Let us consider that the following two requests are issued one after the other.

1. UPDATE (Name=HSIAO)) <Salary, newval=oldval+25>
2. RETRIEVE (Name=HSIAO)

The first request is an update request, where (let us assume) the attribute to be modified is a security attribute. In DBC, we handle update requests that modify a security or clustering attribute in the following way. The records which satisfy the associated query are first deleted from the mass memory. Then, each deleted record is updated, and the database command and control processor (DBCCP) determines the new security atom to which the record belongs, or the new cylinder into which it must be inserted (or both). Finally, each deleted record is reinserted into DBC. Now, if the retrieve request shown above is allowed to be executed before the earlier update completes, no record will be retrieved because the update would have caused the deletion of all relevant records. Hence, the execution of the retrieve request must be stayed until the earlier update completes. That is, the update blocks the retrieve request since the retrieve request may be affected by the update request. We define this concept below.

A request issued subsequent to UPDATE1 [UPDATE QUERY1 <ATTRIBUTE1, NEWVAL1=CONSTANT>] or [UPDATE QUERY1 <ATTRIBUTE1, NEWVAL1=f(OLDVAL1)>] or [UPDATE QUERY1 <ATTRIBUTE1, NEWVAL1=f(<ATTRIBUTE2, OLDVAL2>)>], where ATTRIBUTE1 is a security or clustering attribute, _may_ be _affected_ by UPDATE1 if the request is a delete, retrieve or update request, and the query used in the request may be clashing with QUERY1 on all attributes in the two queries other than ATTRIBUTE1.

The determination of new security atoms and clusters due to new security and clustering attributes, is well documented in [7]. We shall not repeat it here.

5.2.3 Requests Being Blocked Perpetually by Updates

Until now, we had indicated the kinds of requests that are blocked because of a prior update request. We now consider a slightly different problem. Consider that a delete (retrieve, update) request Di has been blocked by an update request Ui that was incomplete at the time Di was issued. After Ui completes, we might assume that Di may now be scheduled for execution. However, this may not always be the case. This is because, in the time between the blocking of Di and the completion of Ui, another update request Uj (which is not blocked by Ui) may have started execution and may be incomplete at the time Ui is complete. Now, if Uj also blocks Di, Di cannot be executed until Uj also completes. It is not very hard to think of an example to illustrate the situation. Consider that the following three requests are issued one after the other.

1. UPDATE ((Salary>50) ∧ (Name=MENON)) <Salary, newval=oldval+25>
2. RETRIEVE ((Salary>75) ∧ (Dept=20))
3. UPDATE (Name=MENON) <Dept, newval=oldval+20>

Let us assume that the first update request is not blocked by any prior update request. Hence, it is scheduled for execution. The retrieve request, however, is blocked by the update request because the query associated with the retrieve request contains the attribute Salary (which is the attribute to be modified in the update request) and may be clashing with the query associated with the update request on the attributes Name and Dept (i.e., all attributes except Salary). Therefore, execution of the retrieve request is stayed until the update completes. The second update request is not blocked by the first update request because it does not contain the attribute salary. Hence, it is immediately scheduled for execution. Therefore, by the time the first update request completes, the second update request may be at some stage of execution. The second update request also blocks the retrieve request because the query associated with the retrieve request contains the attribute Dept (which is the attribute to be modified in the second update request) and the query associated with the retrieve request may be clashing with the query associated with the second update request on the attributes Name and Salary. Thus, the retrieve request must be stayed until the second update request is also completed.

There are two ways to handle the above problem. The first method is to delay the execution of the blocked retrieve (delete, update) requests until no more blocking updates exist. That is, in the above example, execute the retrieve request only after both the update rquests have completed execution.

In the second method, the second update request is not executed until the earlier retrieve request has been scheduled for execution. That is, in effect, the second update request is blocked by the earlier retrieve request. Either of the two aforementioned methods may be used in DBC. We will, henceforth, assume that the first method is the one being employed, but there is no reason why the second method may not be employed.

5.3 The Classification of Updates

Let us now briefly recapitulate the process of update in DBC. The database command and control processor (DBCCP) first accesses the SM to retrieve index terms. These index terms are then intersected in the structure memory interface processor (SMIP). Finally, DBCCP is presented with a set of cylinder numbers that contain the records that will satisfy the query in the update request. The relevant records are then retrieved and update. Based on the udpates made, the SM is also updated. Execution of all subsequent requests that may be affected by this update must be stayed until the final updating of the SM is done.

Naturally, the question may be raised as to whether it would not be possible to update the index terms in the SM before the actual execution of the command in the mass memory. That is, is it not possible to update the SM even as index terms (needed to satisfy the query part of the update request) are being retrieved from it? In the following paragraph, we will illustrate, with an example, why it is not always possible to do this.

Consider, once again, the SM as shown in Figure 25. From the SM, we can see that records containing the keyword <Salary, 75> are present in Cylinders 2 and 3, and records containing the keyword <Salary, 50> are present in Cylinders 1 and 4. Let us assume that each cylinder can hold only two records. In Figure 27, we have shown the two records in each of the Cylinders 1, 2, 3, and 4. Each record, as can be seen, has two keywords (<attribute, value> pairs). Consider that the following update request is now issued.

UPDATE ((Salary≥50) ∧ (Name=HSIAO)) <Salary, newval=oldval+25>

After the exectuion of the above update command, the database now looks as shown in Figure 28. The unwary reader might assume, by looking at the update request alone, that the SM should now look as shown in Figure 26. However, it can be easily verified, by looking at the actual records, that the SM should look as shown in Figure 29 (we will assume that the value of the Salary attribute determines the security atom that a record belongs to). To take another

```
Cylinder 1
  Record 1:  (<Salary, 50> , <Name, HSIAO>)
  Record 2:  (<Salary, 50> , <Name, MENON>)

Cylinder 2
  Record 1:  (<Salary, 75> , <Name, HSIAO>)
  Record 2:  (<Salary, 75> , <Name, MENON>)

Cylinder 3
  Record 1:  (<Salary, 75> , <Name, ANNE>)
  Record 2:  (<Salary, 75> , <Name, JOHN>)

Cylinder 4
  Record 1:  (<Salary, 50> , <Name, ANNE>)
  Record 2:  (<Salary, 50> , <Name, JOHN>)
```

Figure 27. A View of the Database before the Update Command

```
Cylinder 1
  Record 1:  (<Salary, 75> , <Name, HSIAO>)
  Record 2:  (<Salary, 50> , <Name, MENON>)

Cylinder 2
  Record 1:  (<Salary, 100> , <Name, HSIAO>)
  Record 2:  (<Salary,  75> , <Name, MENON>)

Cylinder 3
  Record 1:  (<Salary, 75> , <Name, ANNE>)
  Record 2:  (<Salary, 75> , <Name, JOHN>)

Cylinder 4
  Record 1:  (<Salary, 50> , <Name, ANNE>)
  Record 2:  (<Salary, 50> , <Name, JOHN>)
```

Figure 28. A View of the Database after the Update Command [UPDATE ((Salary>50) ∧ (Name=HSIAO)) <Salary, newval=oldval+25>]

Keyword	Set of indices
<Salary, 50>	(1,1),(4,1)
<Salary, 75>	(1,2),(2,2),(3,2)
<Salary,100>	(2,3)
.	.
.	.
.	.

[UPDATE ((Salary ≥ 50) ∧ (Name = HSIAO)) <Salary, newval = oldval+25>]

Figure 29. A View of the SM after it has been Updated Following the Command

example, consider that instead of the above update request, the following update request is issued.

UPDATE ((Salary$\geq$50) $\wedge$ (Name=ANNE)) <Salary, newval=oldval+25>

After the execution of the above request, the database will now look as shown in Figure 30. Also, the SM should be correctly updated to that shown in Figure 31. In both the examples shown above, the update commands themselves could give us no clue as to what the state of the SM should be after execution of the update commands. Only an actual examination of the records could provide us with this information. Thus, it is impossible to update SM on the basis of the syntax of the update commands alone.

There are, however, certain special circumstances under which we can guess at the new state of SM without actually executing the commands. These situations are listed below.

(1) If the attribute to be modified is not part of a directory keyword. In this case, no change is necessary to SM.

(2) If the attribute to be modified is part of a directory keyword, and *it is not a security or a clustering* attribute, and the query part of the update request contains 'ALL', and the modifier is of TYPE 0 (<A, newval = constant>), or TYPE I (<A, newval = f(oldval)>). In this case, SM is updated as follows. Let the attribute to be modified be A. Then, look through the SM and find all keywords of the form <A, Vi>. Let there be n such keywords, <A, V1>, <A, V2>, ..., <A, Vn>. If the modifier is of TYPE 0, then replace all these keywords by the keyword <A, constant> (Figure 32a illustrates what we mean). If the modifier is of TYPE I, then modify these keywords in SM to <A, f(V1)>, <A, f(V2)>, ..., <A, f(Vn)> (Figure 32b illustrates what we mean).

(3) If the attribute A to be modified is part of a directory attribute, and it is not a clustering or a security attribute, and the query part of the update request consists of a single predicate using the attribute A, and the modifier is of TYPE 0, say <A, newval=constant> or TYPE I, say <A, newval = f(oldval)>. In this case, SM is updated as follows. Look through SM and find all keywords of the form <A, Vi> which satisfy the query conjunction in the query part of the update request. Let there be m such keywords <A, V1>, <A, V2>, ..., <A, Vm>. If the modifier is of TYPE I, then modify these keywords in SM to

```
Cylinder 1
  Record 1:  (<Salary, 50> , <Name, HSIAO>)
  Record 2:  (<Salary, 50> , <Name, MENON>)

Cylinder 2
  Record 1:  (<Salary, 75> , <Name, HSIAO>)
  Record 2:  (<Salary, 75> , <Name, MENON>)

Cylinder 3
  Record 1:  (<Salary, 100> , <Name, ANNE>)
  Record 2:  (<Salary,  75> , <Name, JOHN>)

Cylinder 4
  Record 1:  (<Salary, 75> , <Name, ANNE>)
  Record 2:  (<Salary, 50> , <Name, JOHN>)
```

Figure 30. A View of the Database after the Update Command

[UPDATE ((Salary≥50) ∧ (Name=ANNE))
<Salary, newval=oldval+25>]

Keyword	Set of indices
<Salary, 50>	(1,1),(4,1)
<Salary, 75>	(2,2),(3,2),(4,2)
<Salary,100>	(3,3)
•	•
•	•
•	•

[UPDATE ((Salary > 50) ∧ (Name = Anne)) <Salary, newval=oldval+25)]

Figure 31. A View of the SM after it has been Updated Following the Command

```
Cylinder 1
  Record 1:  (<Salary, 50> , <Name, HSIAO>)
  Record 2:  (<Salary, 50> , <Name, MENON>)

Cylinder 2
  Record 1:  (<Salary, 75> , <Name, HSIAO>)
  Record 2:  (<Salary, 75> , <Name, MENON>)

Cylinder 3
  Record 1:  (<Salary, 100> , <Name, ANNE>)
  Record 2:  (<Salary,  75> , <Name, JOHN>)

Cylinder 4
  Record 1:  (<Salary, 75> , <Name, ANNE>)
  Record 2:  (<Salary, 50> , <Name, JOHN>)
```

Figure 30. A View of the Database after the Update Command

[UPDATE ((Salary≥50) ∧ (Name=ANNE)) <Salary, newval=oldval+25>]

Keyword	Set of indices
<Salary, 50>	(1,1),(4,1)
<Salary, 75>	(2,2),(3,2),(4,2)
<Salary,100>	(3,3)
.	.
.	.
.	.

[UPDATE ((Salary > 50) ∧ (Name = Anne)) <Salary, newval=oldval+25)]

Figure 31. A View of the SM after it has been Updated Following the Command

Keyword	Set of indices
<A,20>	(1,2),(5,1)
<A,30>	(6,3),(2,8)
<A,40>	(1,9)
⋮	⋮

Before

Keyword	Set of indices
<A,75>	(1,2),(5,1),(6,3),(2,8),(1,9)
⋮	⋮

After

Figure 32a. A View of the SM before and after the Update Command [UPDATE ALL <A, newval = 75>]

AD-A094 408 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
DESIGN AND ANALYSIS OF UPDATE MECHANISMS OF A DATABASE COMPUTER--ETC(U)
JUN 80 D K HSIAO, M J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-80-3 NL

2 OF 2
AD A
094408

END
DATE
FILMED
2-81
DTIC

Keyword	Set of indices
<A,20>	(1,2),(5,1)
<A,30>	(4,3),(2,8)
<A,40>	(1,9)

Before

Keyword	Set of indices
<A,40>	(1,2),(5,1)
<A,60>	(4,3),(2,8)
<A,80>	(1,9)

After

Figure 32b. A View of the SM before and after the Update Command

[UPDATE ALL <A, newval = oldval x 2>]

<A, f(V1)>, <A, f(V2)>, ..., <A, f(Vm)>. If the modifier is of TYPE 0, then replace all these keywords by the keyword <A, constant>.

We shall call the above types of update commands as CLASS I, CLASS II, and CLASS III updates, respectively.

We would like to emphasize, at this point, that many update commands are likely to fall into one of the three classes listed above. We feel that, mostly, nondirectory keywords will be updated. Also, even if directory keywords are updated, they are not likely to be security or clustering keywords. This is because, changing a security keyword in a record is likely to change the security atom to which the record belongs. This, in turn, will cause a change in the protection requirement of that record. It is very unlikely, indeed, that a database administrator would want to change the security privileges accorded to a record frequently. Hence, it is very unlikely that an updated attribute is a security attribute. Similarly, changing a clustering keyword of a record could change the cluster that the record belongs to. A cluster contains all records that a user is likely to retrieve together. Therefore, it is very unlikely that a database creator would want to move a record from one cluster to another on a frequent basis. Hence, it is very unlikely that an updated attribute is a clustering attribute. The extra processing involved in order to process an update command which involves the update of a security or clustering attribute should now be clear. If the attribute to be modified is a clustering attribute, then a new cluster has to be calculated for each of the records updated. If a record belongs to a new cluster, then it could possibly mean that the record needs to be placed in a different cylinder. This, of course, involves some overhead. Similarly, if the attribute to be modified is a security attribute, then a new security atom has to be calculated for each record. Once again, this involves some overhead.

We are now in a position to classify the remaining types of updates.

CLASS IV: An update command which does not belong to CLASS I, CLASS II, or CLASS III, and in which the attribute to be modified is part of a directory keyword, but is not a security or clustering attribute.

CLASS V: An update command in which the attribute to be modified is a security attribute but not a clustering attribute.

CLASS VI: An update command in which the attribute to be modified is a clustering attribute but not a security attribute.

CLASS VII: An update command in which the attribute to be modified is both a security and a clustering attribute.

5.4 A Scheme to Determine if Two Queries are Nonclashing

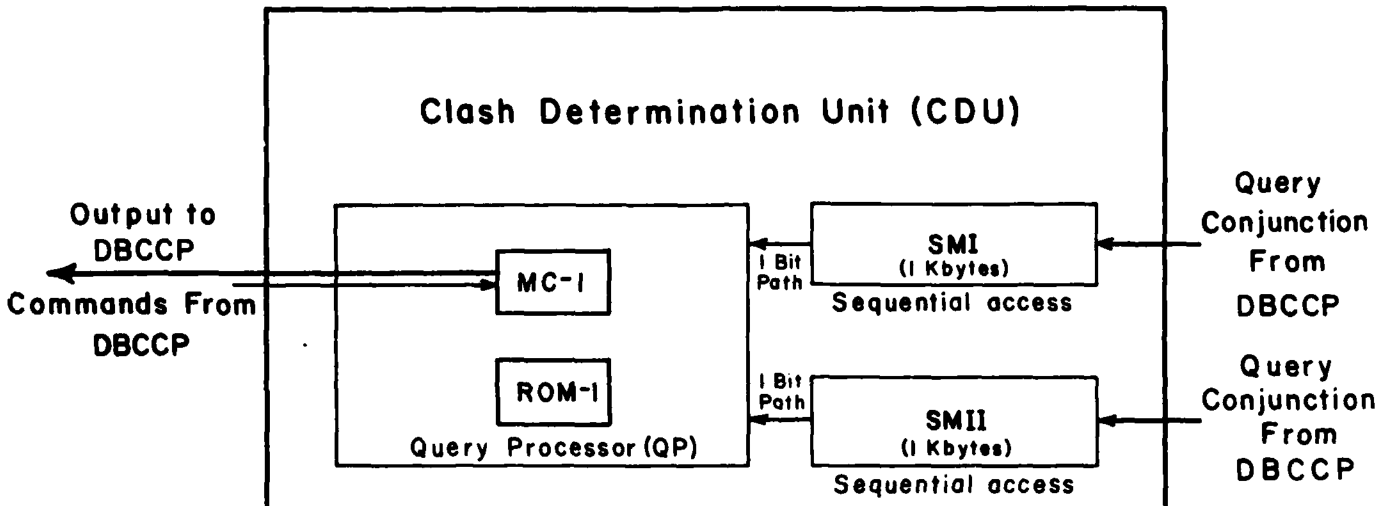
We recall, from our earlier discussion in this section, that the execution of delete, retrieve and update commands must be stayed until the changes caused by a prior update command are reflected in the SM, if these commands are blocked by the earlier update command. To determine if a delete, retrieve or update command may be affected by an earlier update command, one has to determine whether the query of the subsequent delete, retrieve or update command may be clashing with the query of the immediate update command. In this sub-section, we shall propose a hardware scheme to determine if two query conjunctions may be clashing.

The clash determination unit (CDU) is shown in Figure 33. Essentially, it consists of a processor (the query processor (QP)) and two sequential access memories -- SM I and SM II. The sequential memories hold, respectively, the two query conjunctions to be compared. Thus, each sequential memory needs to be no larger than the largest single query conjunction that will be encountered by DBC. This is estimated to be in the neighborhood of 1 Kbytes. CDU utilizes the fact that the predicates in a query conjunction are arranged in ascending order of their attribute identifiers. Each predicate is an <attribute identifier, operator, value> triple. CDU begins to read a sequential stream of bits from both SM I and SM II and to make a bit-serial comparison. Whenever there is a match between the attribute identifier in SM I and the attribute identifier in SM II, the operator and value in SM I are compared with the operator and value in SM II to determine if the query conjunctions are nonclashing on this attribute (See Table I). If the attribute identifier in one of the sequenital memories is larger than the attribute identifier in the other sequenital memory, then the QP skips over the operator and value of the smaller attribute identifier to the next attribute identifier. The above logic is repeated until the conjunctions are found to be nonclashing on some attribute, or it is concluded that the query conjunctions may be clashing on all attributes.

The algorithm executed by the query processor (QP) is presented formally below. It is microcoded in ROM-1 and executed by microsequencer MC-1 (see Figure 33).

ALGORITHM CLASH: Executed by the QP to determine if two query conjunctions are nonclashing or whether they may be clashing.

INPUTS: (1) The first query conjunction in SM I and the number of predicates (M) in it.
(2) The second query conjunction in SM II and the number of predicates (N) in it.



NOTE: The output is a single bit. A '1' indicates the query conjunctions maybe clashing. A '0' indicates that the conjunctions are nonclashing.

Figure 33. A View of the Clash Determination Unit (CDU)

	O1	O2	V2
1	=	=	=V1
2	=	>	<V1
3	=	≥	≤V1
4	=	<	>V1
5	=	≤	≥V1
6	<	=	<V1
7	<	<	anyvalue
8	<	≤	anyvalue
9	<	>	<(V1-1)
10	<	≥	≤(V1-1)
11	≤	=	≤V1
12	≤	<	anyvalue
13	≤	≤	anyvalue
14	≤	>	≤(V1-1)
15	≤	≥	≤V1
16	>	=	>V1
17	>	>	anyvalue
18	>	≥	anyvalue
19	>	<	>(V1+1)
20	>	≤	≥(V1+1)
21	≥	=	≥V1
22	≥	>	anyvalue
23	≥	≥	anyvalue
24	≥	<	≥(V1+1)
25	≥	≤	≥V1

TABLE I. The Set of Conditions Used by the Clash Determination Unit

```
Step 1:   I=1.  J=1.
Step 2:   A1=I-th attribute identifier from SM I.  A2=J-th attribute identifier
          from SM II.  If A1=A2, then go to Step 5.  If A1<A2, then go to
          Step 3.  If A1>A2, then go to Step 4.
Step 3:   I=I+1.  If I>M, then return FLAG=1 and terminate.  Else, go to Step 2.
Step 4:   J=J+1.  If J>N, then return FLAG=1 and terminate  Else, go to Step 2.
Step 5:   [Recall that a query conjunction is a conjunction of predicates,
          where each predicate is a triple of the form (attribute identifier,
          operator, value)].  O1 = I-th operator from SM I.  O2 = J-th operator
          from SM II.  V1 = I-th value from SM I.  V2 = J-th value from SM II.
          Check if any of the 25 conditions listed in Table I hold between O1,
          O2, V1 and V2.  If so, then I=I+1, J=J+1, and go to Step 2.  Else,
          set FLAG=0 and terminate.  [If any of the conditions listed in Table
          I hold, we conclude that the query conjunctions may be clashing on
          this particular attribute].
```

[Note: The result of this algorithm is returned in a variable called FLAG. FLAG=0 indicates that the query conjunctions are nonclashing. FLAG=1 indicates that the query conjunctions may be clashing.]

Earlier, we had indicated that it would be necessary to decide if two queries were nonclashing or if they may be clashing. The CDU only determines if two query conjunctions are nonclashing or may be clashing. In order to make use of this facility, and for other reasons [1-6], we do the following in DBC. Each delete (retrieve, update) command is split up into m delete (retrieve, update) commands, where m is the number of conjunctions in the query associated with the delete (retrieve, update) command. Thus, the command

DELETE (Name = HSIAO) $\vee$ (Salary > 50)

is split up into two commands as follows.

DELETE (Name=HSIAO)

DELETE (Salary>50)

Now, all delete, update and retrieve commands utilize only query conjunctions. All our earlier discussion on commands with queries is equally applicable to commands with query conjunctions, since a query conjunction is also a query. Thus, we have reduced the problem of trying to determine if two queries are nonclashing (or may be clahsing) to the problem of trying to determine if two query conjunctions are nonclashing (or may be clahsing). The CDU may be readily utilized to serve the above purpose.

5.5 DBCCP Processing

In this sub-section, we will consider the kinds of processing done by DBCCP in order to execute any given request issued by a user. A description of some of the data structures that will have to be used by DBCCP in order to do the processing is also presented. Since the processing needed for delete and retrieve commands is different from the processing needed for update commands, we shall describe these in different sub-sections.

5.5.1 Data structures Needed for DBCCP Processing

DBCCP will need the use of two tables to process the various kinds of requests issued to it. We will describe each in turn.

A. Incomplete Updates Table (IUT)

There is only one such table in DBC. Each entry in IUT is an update command on which execution has been started but not completed. That is, none of the update commands in this table were blocked by any other update command and hence were scheduled for execution. Each of the update commands in IUT will be in one of the following stages of execution.

(1) An order has been issued to the SM to retrieve index terms corresponding to the command, but the intersected index terms have still not been obtained by DBCCP.

(2) The intersected index terms corresponding to the command have been obtained by DBCCP, and orders have been issued to the mass memory (MM) to begin execution. However, the MM has not yet finished executing the orders.

Basically, none of the updates in IUT are blocked or complete. An update is completed only when the SM has been modified to reflect the changes caused by the update. Each entry in IUT consists of three fields. Field 2 will store the modifier part of the update command. Field 3 will contain a unique number, called an update number, that is issued to each update command by DBC. Field 1 needs more explanation. Recall that update commands issued to DBC are split up into m separate update commands, where m is the number of conjunctions in the query associated with the original update command. Each newly created update command has an associated query conjunction. Field 1 stores the query conjunction associated with the update command minus the predicate (if it exists) containing the attribute to be modified. An example should serve to calrify the situation. If the update command is UPDATE ((Salary>50) ∧ (Name=HSIAO)) <Salary, newval=oldval+25>, then Field 1 will store the conjunction (Name=HSIAO). The predicate (Salary>50) is removed from the conjunction since Salary is the attribute to be modified. Note that the removal of the predicate containing the attribute to be modified from the query conjunction could cause the query conjunction to become null. In this case, a null value is stored in Field 1. A logical view of IUT is shown in Figure 34. Any retrieve, delete or update request received by DBC may be affected by one or more of the updates in IUT and hence may be blocked until the update(s) are removed from IUT after its (their) completion.

B. The Blocked Requests Table (BRT)

There is one such table in DBC. Each entry in the blocked requests table (BRT) contains a request which is blocked owing to one or more updates in IUT. These requests may be delete, retrieve or update requests. Each entry in BRT consists of two fields. Field 1 will contain the actual request itself (in its entirety). Field 2 will contain a list of update numbers corresponding to updates which cause this request to be blocked. A logical view of such a table is shown in Figure 35. It indicates that the delete request

DELETE (Salary> 50)

is blocked by updates with update numbers 1, 5 and 12. All these three updates must be completed before the delete request can be scheduled for execution.

5.5.2 The Handling of Retrieve and Delete Requests

When a retrieve (or delete) request is received by DBCCP, it is first split up into m retrieve (or delete) commands, where m is the number of conjunctions in the query associated with the retrieve (or delete) command. The following is done for each of the m commands, D1, D2, ..., Dm, thus generated. We shall explain the process for command Di which has the query conjunction Qi associated with it. Each entry in IUT is accessed in turn. If the attribute, then ODU is invoked to determine if Qi clashes with the conjunction in Field 1 of the entry. Else, a check is made to see if the attribute to be modified in Field 2 of the entry is part of the conjunction Qi. If it is not, then we conclude that the update cor esponding to the entry does not block request Di. If the attribute to be modified in Field 2 of the entry is part of the query conjunction Qi, then CDU is invoked to determine if Qi clashes with the conjunction in Field 1 of the entry. If the conjunction in Field 1 is null, we may conclude, without invoking CDU, that Qi clashes with this null entry. In this way, each entry in IUT is accessed and the update numbers of all updates that block the request Di are determined. If no update that blocks the request Di is found, then Di is scheduled for execution. Otherwise, a new entry is made in BRT with the request Di in Field 1 of the entry, and a list of update numbers of all blocking updates in Field 2 of the entry.

Query Conjunction	Modifier	Update Number
(Salary>50)	<Salary,newval= oldval/2>	1
•	•	•
•	•	•
•	•	•

Figure 34. A Logical View of the Incomplete Updates Table (IUT)

Request	List of Update Numbers
[Delete (Salary > 50)]	1,5,12
•	•
•	•
•	•

Figure 35. A View of the Blocked Requests Table (BRT)

Each time an update request issued to DBC completes execution, BRT is accessed, and the update number corresponding to the just completed update request is removed from all lists (in the second fields of BRT) in which it is a member. If any of the second fields in BRT is thus caused to become an empty list, the corresponding request Di in Field 1 is examined again to see if any new update request (which was issued subsequent to Di) in IUT is causing Di to be blocked. If not, Di is scheduled for execution and the entry corresponding to Di is removed from the BRT. However, if Di is being blocked by some new updates in the IUT, then Field 2 of the entry corresponding to Di in BRT is filled with a list of update numbers of all blocking updates.

Thus, a retrieve or delete request, received by DBCCP, is scheduled for execution immediately if no incomplete update caused its blockage. Otherwise, the execution of the retireve or delete request is stayed until such times no blocking updates remain. The process is described algorithmically below.

ALGORITHM DELRET: Executed by DBCCP on receipt of a delete or retireve request.

Step 1: Split the request (in the manner indicated in an earlier sub-section) into m separate requests, where m is the number of query conjunctions in the query associated with the request. Each newly created command has an associated query conjunction. Let the newly created commands be D1, D2, .., Dm and let the associated query conjunctions be Q1, Q2. ..., Qm. Set i=1. In Steps 2 through 6, we process the i-th command.

Step 2: [Let there be N entries in the IUT.] Set j=1. Set L-null list.

Step 3: Access the j-th entry in the IUT. Let the attribute to be modified in Field 2 of the j-th entry be AT. If AT is a security or clustering attribute, then go to Step 4. Else, check to see if AT is part of the query conjunction Qi. If not, then go to Step 5. Else, go to Step 4.

Step 4: Place Qi in SM I of CDU. Place the conjunction in Field 1 of the j-th entry of the IUT in SM II of CDU. Invoke Algorithm CLASH to determine if the two query conjunctions are nonclashing. [If the conjunction in Field 1 is null, we conclude, without invoking the CDU, that the two query conjunctions may be clashing.] If the query conjunctions may be clashing, then add the update number in Field 3 of the j-th entry in the IUT to L and go to Step 5. If the two query conjunctions are nonclashing, then go to Step 5. [Note that in L we are building up a list of update numbers of blocking updates.]

Step 5: j=j+1. If j<=N, then go to Step 3. If j>N, then check to see if L is equal to the null list. If it is, then go to Step 6. If L is nonnull, then create a new entry in the BRT with Di in Field 1 and L in Field 2. Go to Step 7.

Step 6: Schedule the execution of request Di and go to Step 7.

Step 7: i=i+1. If i>m, then terminate. Else, go to Step 2.

5.5.3 The Handling of Update Requests

When an update command is received by DBCCP, it is first split up into m update commands, where m is the number of conjunctions in the query associated with the update request. The following is done for each of the m commands, U1, U2, ..., Um, thus generated. We shall explain the process for command Ui which has associated with it the conjunction Qi.

The update request Ui is handled essentially in two stages. The processing done by DBCCP in the first stage is similar to the processing done by DBCCP to handle a retrieve or a delete request. Each entry in IUT is accessed and a check is made to see if the attribute to be modified in Field 2 of the entry is a security or clustering attribute. If so, then CDU is invoked to determine if Qi clashes with the conjunction in Field 1 of the entry. Else, a check is made to see if the attribute to be modified in Field 2 of the entry is part of the conjunction Qi. If not, we conclude that the update corresponding to the entry does not block Ui. Otherwise, CDU is invoked to determine if Qi clashes with the conjunction in Field 1 of the entry. If the conjunction in Field 1 is null, we conclude, without invoking CDU, that Qi clashes with the conjunction in Field 1.

In this way, each entry in IUT is accessed and the update numbers of all updates that block the request Ui are determined. If no update that blocks the request is found, the processing enters the second stage and this will be described later. Otherwise, a new entry is made in the blocked request table (BRT) with the request Ui in Field 1 of the entry, and a list of update numbers of all blocking updates in Field 2 of the entry. Ui will be scheduled for execution only after all the blocking updates in Field 2 of the BRT entry have completed and no more blocking updates exist in IUT (see Algorithm COMPLETE in the next sub-section).

When an update request Ui has no updates blocking it, DBCCP enters the second stage of execution. In this stage, the request Ui is scheduled for execution. First, DBCCP tries to determine the class to which the update request Ui belongs. If it belongs to one of CLASS I, CLASS II, or CLASS III, then request Ui is not entered into IUT. This is because the SM can be updated even before Ui is executed, and, hence, Ui will not block any subsequent request. If Ui does not belong to any of these three classes, then it is entered into IUT since it could, potentially, block subsequent requests to DBC. The entry in IUT is as follows. Field 1 will contain Qi minus the predicate (if it exists) containing the attribute to be modified. Field 2 contains the modifier and Field 3 contains an update number which is a unique number given to update request Ui by DBCCP. The entire process is shown algorithmically below.

```
ALGORITHM UPDATE:  Executed by DBCCP on receipt of an update request.
[STAGE 1]

Step 1:   Split the request (in the manner indicated in an earlier sub-section)
          into m update requests, where m is the number of query conjunctions
          in the query associated with the request. Each newly created
          command has an associated query conjunction. Let the newly created
          commands be U1, U2, ..., Um and let the associated query conjunctions
          be Q1, Q2, ..., Qm. Set i=1. In Steps 2 through 8, we process the
          i-th command.
Step 2:   [Let there be N entries in the IUT.] Set j=1. Set L=null list.
Step 3:   Access the j-th entry in the IUT. Let the attribute to be modified
          in Field 2 of the j-th entry be A1. Check to see if A1 is a security
          or clustering attribute or is part of the query conjunction Qi.
          If not, then go to Step 5. Else, go to Step 4.
Step 4:   Place Qi in SM I of the CDU. Place the conjunction in Field 1 of the
          j-th entry of the IUT in SM II of the CDU. Invoke Algorithm CLASH
          to determine if the two query conjunctions are nonclashing. [If
          the conjunction in Field 1 of the j-th entry is null, we conclude,
          without invoking the CDU, that the query conjunctions may be clashing.]
          If the query conjunctions may be clahsing, then add the npdate number
          in Field 3 of the j-th entry in the IUT to L and go to Step 5.
          [In L, we are building up a list of blocking updates.]
Step 5:   j=j+1. If J<=N, then go to Step 3. If j>N, then check to see if
          L is equal to the null list. If it is, then go to Step 6. If L is
          nonnull, then create a new entry in the BRT with Ui in Field 1 and L
          in Field 2. Go to Step 9.

[STAGE II]

Step 6:   Let A=attribute to be modified in Ui. If A is not part of a directory
          keyowrd, then go to Step 7. (Ui is a CLASS I request, and no change
          is needed in the SM). Else, if A is not a security or clustering
          attribute and Qi='ALL', then mark the request as belonging to CLASS II
          and go to Step 7. Else, if A is not a security or clustering attri-
          bute and Qi is a single predicate using A, then mark the request as
          belonging to CLASS III and go to Step 7. Else, if A is not a security
          or a clustering attribute, then mark the request as belonging to
          CLASS IV and go to Step 8. Else, if A is a security but not a
          clustering attribute, then mark the reuqest as belonging to CLASS V
          and go to Step 8. Else, if A is a clustering but not a security
          attribute, then mark the request as belonging to CLASS VI and go to
          Step 8. Else, mark the reuqest as belonging to CLASS VII and go to
          Step 8.
Step 7:   Schedule the request for execution, and go to Step 9.
Step 8:   Make an entry in the IUT as follows. Field 1 will contain Qi
          minus the predicate (if it exists) containing attribute A. Field 2
          will contain the modifier part of Ui. Field 3 will contain the update
          number assigned to request Ui. Go to Step 7.
Step 9:   i=i+1. If i>m, then temrinate. Else, go to Step 2.
```

5.5.4 DBCCP Processing on Completion of an Update Request

We now describe the algorithm that will be executed by DBCCP when it is notified by MMC that an update request has been completely executed by MM.

ALGORITHM COMPLETE: Executed by DBCCP when notified of the completion of an update request by MMC.

```
INPUTS:   (1)  The update number (UN) of the completed request.
          (2)  A set of index terms that must be modified in SM.
Step 1:   Search through the IUT and remove the entry which contains UN in
          Field 3.
Step 2:   Request the SM to make the necessary modifications indicated by the
          second input.
Step 3:   For each entry in the BRT, do Steps 4 and 5.
Step 4:   Check if UN is one of the numbers in the list in Field 2 of the entry.
          If so, then delete UN from the list in Field 2.  If the removal of
          UN from the list in Field 2 causes the list to become null, then go
          to Step 5.  Else, do nothing.
Step 5:   [If possible, schedule the request in Field 1 of the entry for
          exectuion.]  If it is a delete or retrieve request, then invoke
          Algorithm DELRET.  If it is an update request, then invoke
          Algorithm UPDATE.
Step 6:   Terminate.
```

5.6 Command Execution in the Mass Memory (MM)

In this sub-section, we would like to describe the actual execution of the update request in MM after they have been scheduled for execution by DBCCP. We will begin by assuming that TIPs have small buffers attached to them. Later, we shall show the improvement in performance that can be obtained by having track-size buffers with each TIP. Finally, we shall propose a way of further improving the throughput of DBC. By referring back to the analytical model developed in Section 2, we shall arrive at some quantitative figures for the data loop throughput.

5.6.1 Calculating the Number of Revolutions for an Update

Let us assume that each TIP has a buffer that is only big enough to hold one record. Therefore, only one record can be stored and updated in one revolution of the disk device. Essentially, the process of update is as follows. In the first revolution, the first record (in each track) that satisfies the given query conjunction is stored in the corresponding TIP buffer. Each TIP then updates the record in its buffer. In the second revolution, the updated record is written back onto the track and the second record that satisfies the query conjunction is read out into the TIP buffers from each track. In this way, if n records have to be updated in a track, the corresponding TIP will take n+1 revolutions to do the update. Consider a typical example, where the size of a track is 30 Kbytes, and the average size of a record is 200 bytes. This means, that there are about 150 records in a track. Assuming that 10% of the records in a track will be updated, we see that the process of updating will take 16 revolutions.

Consider, instead, that each TIP has a track-size buffer associated with it. In the first revolution, all records that satisfy the query conjunction associated with the update command are read into the buffers by the TIPs. In the second revolution, all the records are updated by the TIPs. The third revolution may then be used to write back all the updated records into the TIP buffers. Thus, only three revolutions are needed to perform an update. Actually, we can do a bit better than this. Since each TIP really consists of a disk interface processor (DIP) and a control interface processor (CIP), the first revolution may be utilized for both reading and updating of records. That is, even as the DIP is reading a record and placing it in the buffer, the CIP is updating a previously read record in the buffer. However, since a record cannot, probably, be updated as fast as it can be read off the tracks, we will assume that the process of update still takes three revolutions to execute.

5.6.2 A Modification

We will now suggest a modification to the above method of doing updates in order to get some additional improvement. The method we use takes only two revolutions per update, instead of three. A set of orders on a particular cylinder is executed as follows. The first revolution is devoted to reading all the records into the track-size buffer associated with each TIP. Subsequent orders (that are within this set of orders) will not read the records from the track. Instead, they will read the records from the track-size buffer. We recall that this track-size buffer is a sequential memory. Let us refer to the sequential reading of all the records in the buffer, from end to end, as a <u>readthrough</u> of the buffer. If the shift rate of the sequential memory is greater than the readout rate of the track device, requests will be handled quicker than before because the time taken to read through the entire track-size buffer (one readthrough) is less than the time taken to read an entire track of information off the disk (one revolution).

Remember, that each of the TIPs utilizes a bit map to remember the positions of the records which have been deleted. In order to handle update requests, we shall utilize another bit map to remember the positions of the records which were found to satisfy the query conjunction associated with the update request. Thus, each record in the buffer has two bits associated with it. The first bit is the *delete bit*, and the setting of this bit indicates that the corresponding record has to be deleted. The second bit is the *update bit*, and its use will be indicated in the following paragraph.

The execution of an update request proceeds as follows. The DIP begins to read the records from the buffer and it sets the update bits of all records that satisfy the query conjunction associated with the update request. CIP also reads the records in the buffer and updates those records whose update bits have been set. DIP and CIP operate in a pipeline fashion, that is, even as DIP is reading a record and checking it for satisfaction with the query conjunction, CIP is updating a record that has been previously read from the buffer by the DIP. The algorithms executed by the CIP and the DIP are indicated below.

ALGORITHM DIP: Executed by the DIP in response to an update request.

INPUTS: (1) The records in the track-size buffer.
(2) The query conjunction part of the request stored in the query buffer.

Step 1: For each undeleted record in the buffer, do Step 2.

Step 2: Check to see if the record satisfies the query conjunction in the query buffer (by sequential reading of the record from the track-size buffer and of the query conjunction from the query buffer). If so, then set the corresponding update bit. Indicate to the CIP that it has finished processing the record.

Step 3: Terminate.

ALGORITHM CIP: Executed by the CIP in response to an update request.

INPUTS: (1) The records in the buffer.
(2) The modifier part of the update request.

Step 1: For each undeleted record in the buffer, do Steps 2 and 3.

Step 2: Wait until the DIP indicates (via the communication area) that it has finished processing the segment to which the record belongs.

Step 3: Check the update bit corresponding to the record. If it has been set, then update the record using the modifier part of the update request and then reset the update bit. If the update bit has not been set, then do nothing.

Step 4: Terminate.

It is difficult to approximate the time taken to perform an update, since the time taken to update a record depends upon the complexity of the update (e.g., whether it involves a multiplication or an addition) and the number of records that must be updated. However, it is felt that an update request can be completed in two revolutions of the disk device. One additional revolution will, of course, be needed at the end of a set of orders in order to write back all the records from the buffer onto the track. The writing back of all the records in the buffer may be done in the same revolution that is used to insert records into the track (see Section 4 on compaction).

Finally, we make use of the analytical model developed in Section 2 in order to make a quantitative estimate of the performance improvements obtained as a result of the suggested change. Figures 36a and 36b show the results, in graphical form, of a comparison study of throughput rates with and without the suggested changes. The improvements (especially in the case where 50% of the requests are update requests), as can be seen, are quite gratifying.

5.7 The Handling of the Various Request Classes

We have already classified the update reauests into seven different classes. Also, in an earlier sub-section, we have indicated how DBCCP determines the class to which a given update request belongs. By the time the update request is received for execution by MM, the class of the request is indicated in the request itself. Algorithm CIP and Algorithm DIP, which were presented in the previous sub-section, are the algorithms executed by MM in response to update requests that belong to one of CLASS I, CLASS II or CLASS III. That is, there is no difference in the manner in which CLASS I, CLASS II or CLASS III reauests are handled by MM (these requests are handled differently by the DBCCP and they reauire different kinds of updates in the SM). However, if the request belongs to CLASS IV, CLASS V, CLASS VI or CLASS VII, MM will have to execute a different set of algorithms. We shall consider each class in turn.

5.7.1 Handling CLASS IV Requests

We recall that a CLASS IV update request is one which does not belong to CLASS I, CLASS II or CLASS III, and in which the attribute to be modified is part of a directory keyword, but is not a security or a clustering attribute. In this case, we need to modify SM after MM completes its update part. This means that the entry corresponding to the old value of the attribute to be modified must be deleted from the SM and an entry corresponding to the new value of that attribute must be inserted into the SM. This, in turn, means that TIPs have to keep track of the keywrods that were modified in the records in MM. In order to do this, TIPs use the following tables.

(1) The tagged deletion table (TDT).

(2) The untagged deletion table (UDT).

(3) The structure memory insertion table (SMIT).

We have already described TDT and UDT in Section 4. SMIT is similar to the other two tables and is shown in Figure 37. As can be seen, each entry in it consists of two fields. The first field contains a directory keyword. The second field contains a set of pairs, where each pair is of the form (cylinder number, security atom number). Entries in SMIT must later be inserted into SM.

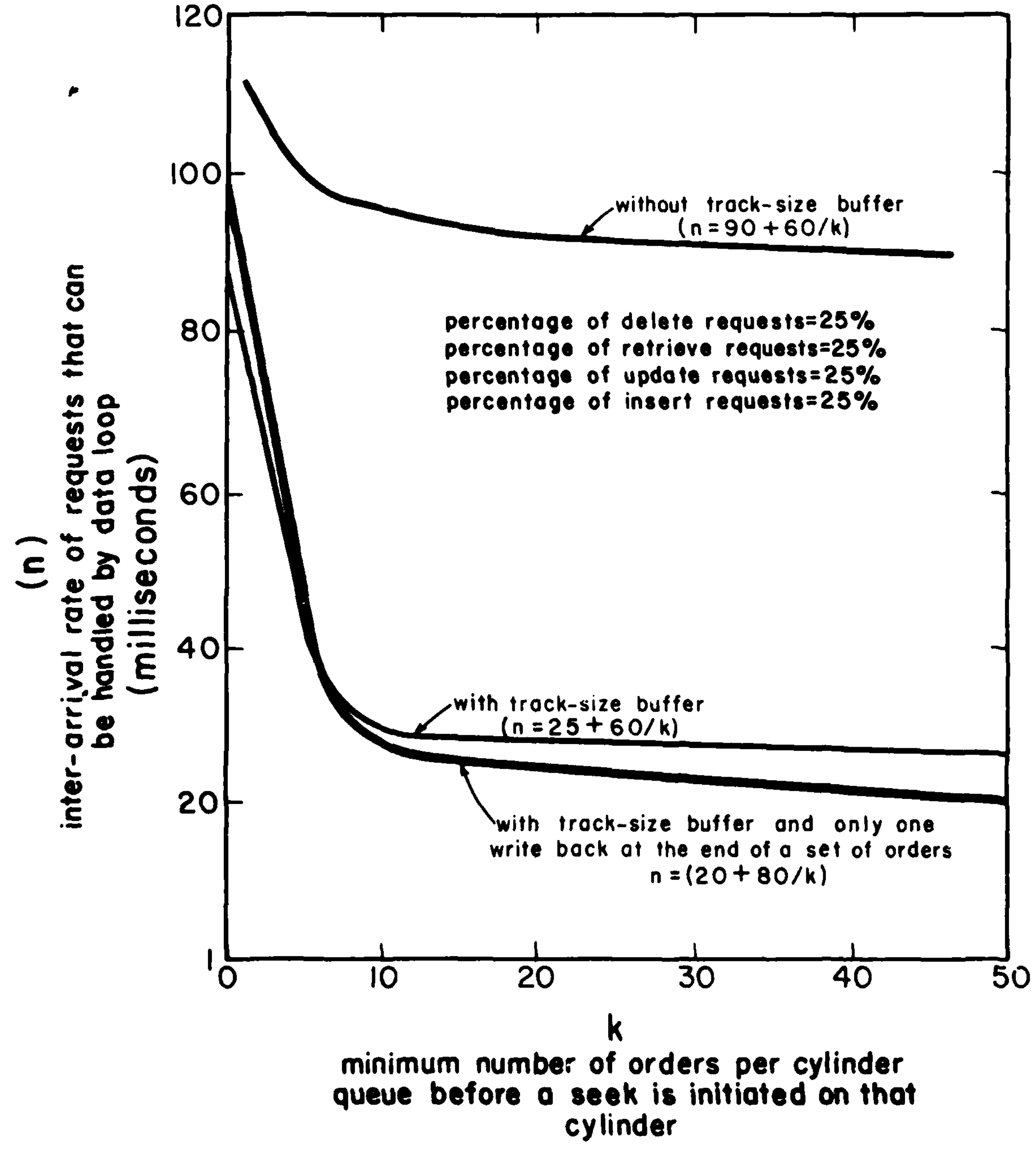


Figure 36a. Graph Showing Results of Study when Percentage of Update Requests is 25%

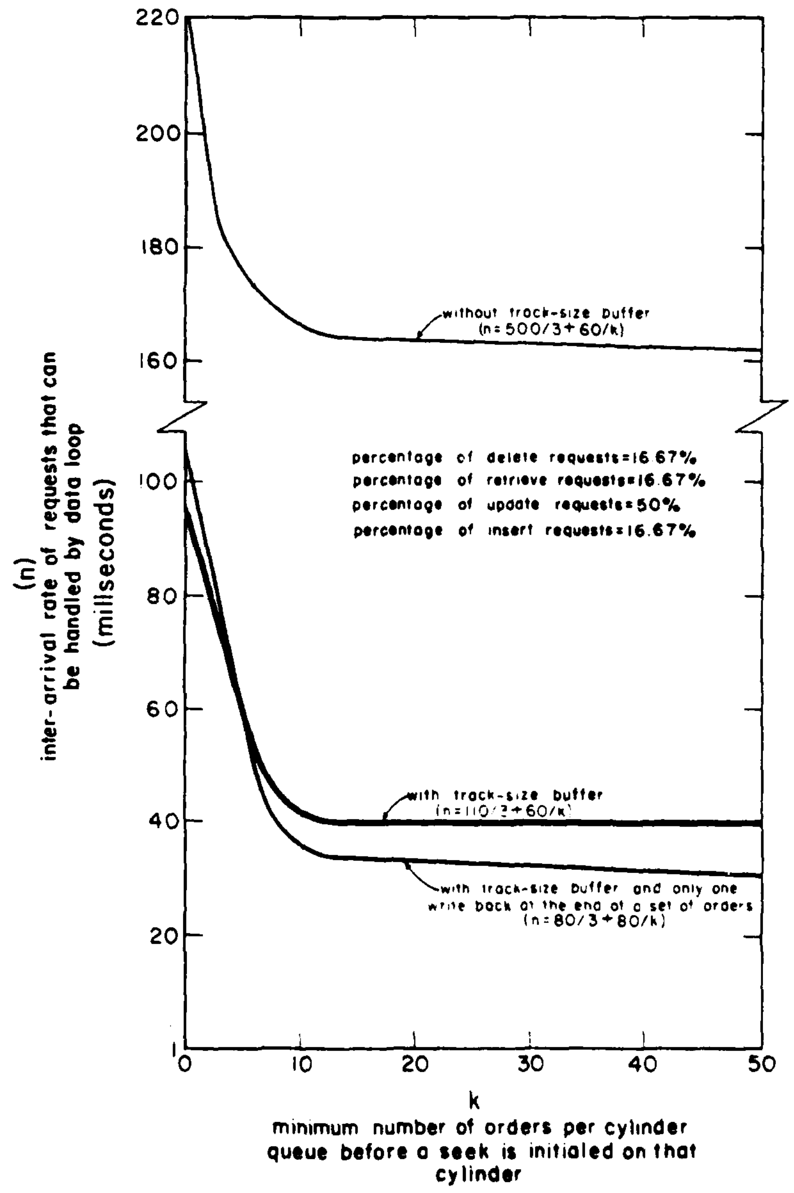


Figure 36b. Graph Showing Results of Study when Percentage of Update Requests is 50%

Directory Keyword	A set of pairs. Each pair is of the form (cylinder number, security atom number)
•	
•	
•	

Figure 37. The Structure Memory Insertion Table (SMIT)

We are now in a postion to indicate the algorithms executed by CIP and DIP to handle update requests of CLASS IV. DIP executes exactly the same algorithm as it does to handle CLASS I, CLASS II and CLASS III requests, and which we have earlier called Algorithm DIP. CIP, however, executes a different algorithm.

ALGORITHM CIP IV: Executed by CIP in response to an update request of CLASS IV.

INPUTS: (1) The records in the sequential buffer.
(2) The modifier part of the update request.
(3) The TDT and SMIT in the RAM buffer.

Step 1: For each undeleted record in the buffer, do Steps 2 through 5.

Step 2: Wait until the DIP indicates (via tne communication area) that it has finished processing the segment to which the record belongs.

Step 3: Check the update bit Corresponding to the record. If it has not been set, then do nothing. Else, go to Step 4.

Step 4: Find the keyword, in the record, corresponding to the attribute to be modified. Let the attribute to be modified be A1, and let the keyword corresponding to this attribute be <A1, V1>. Also let the record belong to Security Atom S, and let the record belong to Cylinder M. Search through the TDT, looking for an entry with the keyword <A1, V1> in Field 1. If no such entry exists, create a new entry with the keyword <A1, V1> in Field 1, and the pair (M,S) in Field 2. If such an entry exists, look through the list in the second field of the entry to see if the pair (M,S) is part of the list. If not , add the pair (M,S) to the list in Field 2. Go to Step 5. [Here, we are putting the SM entry corresponding to the old value of attribute A1 into the TDT, since this entry may have to be deleted from the SM.]

Step 5: Perform the necessary update using the modifier part of the update command. Let the new value of the attribute to be modified (A1) be V2. Then, the new keyword corresponding to the attribute A1 is <A1, V2> . Search through the SMIT, looking for an entry with the keyword <A1, V2> in Field 1. If no such entry exists create a new entry with the keyword <A1, V2> in Field 1, and the pair (M,S) in Field 2. If such an entry exists, look through the list in the second field of entry to see if the pair (M,S) is part of the list. If not, add the pair (M,S) to the list in Field 2. Reset the update bit corresponding to this record. [In this step, we are putting an entry corresponding to the new value of attribute A1 into the SMIT. This entry may later have to be inserted into the SM. Nothice that this new enrty may later have to be inserted into the SM. Notice that this new entry will not have to be inserted into the SM if a later update causes further modification of attribute A1 in this record.] Reset the update bit.

Step 6: Terminate.

Thus, we see that the execution of update requests belonging to CLASS IV will cause entries to be made in TDT and SMIT. Earlier in Section 4, we had shown how the process of compaction causes entries to be made in TDT and UDT. Thus, at the end of a set of orders, each TIP has three tables to be transferred to the mass memory controller (MMC). These are TDT, UDT and SMIT. In Section 4, we had also indicated the algorithm that will be executed by MMC in order to form the structure memory deletion table (SMDT) from TDT and UDT. Here, we shall indicate the MMC algorithm to form SMIT.

ALGORITHM SMIT: Executed by MMC to form the structure memory insertion table (SMIT).

INPUTS: One SMIT per TIP.

Step 1: Concatenate all the SMITs into a single SMIT. Let there be N entries in the newly formed SMIT. For each entry in the newly created SMIT, do Steps 2 and 3.

Step 2: [Assume that we are processing the I-th entry in SMIT.] Let the entry be of the form <K,S>, where S is a set of pairs and K is a directory keyword. Look through every entry below this entry (i.e., entries I+1 through N) and see if any of these entries below it have the value K in Field 1. Put all these entries, below the I-th entry, with value K in Field 1, into a set Q. For each entry in set Q, do Step 3.

Step 3: Let the entry be of the form <K, R>, where R is a set of pairs. Delete this entry from the SMIT and change the I-th entry in the SMIT to <K, S U R>.

Step 4: Terminate.

Thus, at the end of the execution of a set of orders, MMC obtains a SMDT and a SMIT. MMC sends these tables to DBCCP, which then instructs SM to first insert the index terms in SMIT and then delete the index terms in SMDT.

5.7.2 Handling CLASS V, CLASS VI and CLASS VII Requests

A CLASS V update request is one in which the attribute to be modified is a security attribute but is not a clustering attribute. A CLASS VI update request is one in which the attribute to be modified is a clustering attribute but is not a secuirty attribute. A CLASS VII update request is one in which the attribute to be modified is both a security and a clustering attribute. The TIPS execute the same algorithm to handle an update request in one of these three classes.

DIP executes the same algorithm as it does to handle CLASS I, CLASS II, CLASS III and CLASS IV requests (which we have earlier called Algorithm DIP). CIP, however, executes a different algorithm.

ALGORITHM CIP V: Executed by the CIP in response to an update request of CLASS V, CLASS VI or CLASS VII.

INPUTS: (1) The records in the sequential memory.
(2) The modifier part of the update request.
(3) The TDT in the RAM buffer.

Step 1: For each undeleted record in the buffer, do Steps 2 through 5.

Step 2: Wait until the DIP indicates (via the communication area) that it has finished processing the segment to which the record belongs.

Step 3: Check the update bit corresponding to the record. If it has not been set, then do nothing. Else, go to Step 4.

Step 4: Find the keyword, in the record, corresponding to the attribute to be modified. Let the attrbute to be modified be A1, and let the keyword corresponding to A1 be <A1, V1> . Also, let the record belong to Security Atom S and Cylinder M. Search through the TDT, looking for an entry with the keyword <A1, V1> in Field 1. If no such entrv exists, create a new entry with the keyword <A1, V1> in Field 1, and the pair (M,S) in Field 2. If such an entry exists, look through the list in the second field of the entry to see if the pair (M,S) is part of the list. If not add the pair (M,S) to the list in Field 2. Go to Step 5. [Here, we are putting into the TDT, the SM entry corresponding to the old value of attribute A1, since, this entry may have to be deleted form the SM.]

Step 5: Perform the necessary update using the modifier part of the update command. Set the delete bit corresponding to this record. Send the record over to the MMC. [Setting the delete bit will caus the removal of all those entries in the SM which point to this record. This removal will happen during the process of compaction.] Reset the update bit.

Step 6: Terminate.

Essentially, CIP updates the records that satisfy the query. I then marks these records as having been deleted and sends these records over to MMC which in turn sends them over to DBCCP. For each record that is received by it, DBCCP does the following.

(1) If the update request is of CLASS V, it determines a new securitv atom for the record.

(2) If the update request is of CLASS VI, if determines a new cluster and cylinder for insertion.

(3) If the update request is of CLASS VII, it determines a new securitv atom, a new cluster and a new cylinder for insertion.

Then, it issues an insert command with that record as argument. The insert command will, of course, cause many new index terms to be created in the SM (one for each directory keyword in the record). The old index terms associated with the record will be removed by the process of compaction which takes place at the end of a set of orders.

The flow of seven classes of update commands in DBC is depicted in Figure 38. In this figure, we also name the algorithms needed for the execution of each class. Attached to the left of the algorithms named are the tables necessary for supporting the running of the algorithm. Attached to the right of algorithms named are the hardware components required for running the algorithm and utilizing the tables. This figure gives an overall structure of the information, algorithms and components needed for update.

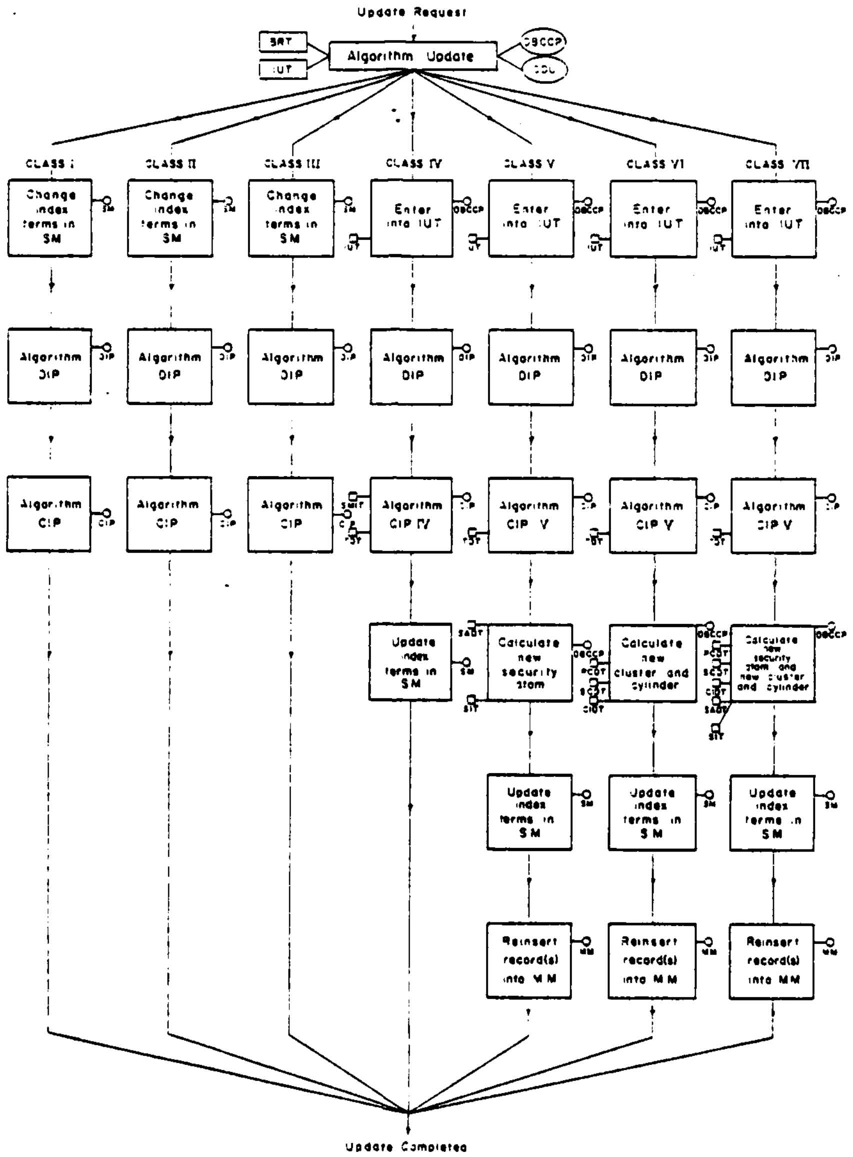


Figure 38. The Command Flow of Seven Classes of Update

6. SUMMARY OF ARCHITECTURAL ENHANCEMENTS

In the preceding sections, we considered each of the processes of insertion deletion and update in isolation. In this section, we propose to present an integrated picture of DBC with all the enhancements that have been suggested in the previous sections.

A set of orders on a particular MAU is handled in DBC as follows. In the first revolution of the disk device, TIPs will read all the records into the track-size buffers which are associated with each TIP. In the subsequent revolutions, retrieve, delete, insert and update commands are executed as described in the following sub-sections.

6.1 <u>Handling Retrieves</u>

If the request is a retrieve request, DIP will begin to search the records in the buffer, from end to end, to find those that satis y the query conjunction in the given request. DIP will set the update bits corresponding to all qualitying records. Simulataneously, CIP also begins to search the records in the buffer. Whenever CIP encounters a record whose update bit has been set, it transmits the record from the buffer to MMC and then resets the update bit corresponding to the record. CIP and DIP can operate in a pipeline fashion, since the buffer consists of many individually accessible segments. Therefore, when DIP is looking to see if a particular record in a segment qualifies for retrieval, CIP can be sending another record (in another segment which DIP has finished processing) to MMC. The algorithms are presented formally below.

```
ALGORITHM DIPRETRIEVE:  Executed by DIP in response to a retrieve request.

INPUTS:   (1)  The records in the track-size buffer.
          (2)  The query conjunction stored in the query buffer.

Step 1:   For each undeleted record in the buffer, do Step 2.
Step 2:   Check to see if the record satisfies the query conjunction in the query
          buffer (by sequential reading of the record from the track-size
          buffer and of the query conjunction from the query buffer). If so,
          then set the corresponding update bit. Indicate to CIP that it has
          finished processing the record.
Step 3:   Terminate.

ALGORITHM CIPRETRIEVE:  Executed by CIP in response to a retrieve request.

INPUTS:   (1)  The records in the buffer.
          (2)  The update bit map in the RAM.

Step 1:   For each undeleted record in the buffer, do Steps 2 and 3.
Step 2:   Wait until the DIP indicates (via the communication area) that
          it has finished processing the segment to which the record belongs.
Step 3:   Check the update bit corresponding to the record. If it has been set,
          then send the record over to MMC and then reset the update bit. If
          the update bit has not been set, then do nothing.
Step 4:   Terminate.
```

The time taken to execute the retrieve request is almost equal to the time taken to read through the sequential track-size buffer from end to end (we have earlier called this one readthrough). Since we feel that the shift rate of the sequential buffer will be greater than the readout rate from the tracks, a retrieve can be executed in less than the time taken for one revolution on a disk device. This method of doing retrieval has another advantage over the retrieval method previously proposed for DBC (where the records were read directly off the tracks and compared to a query conjunction 'on the fly'). This may be explained by taking a closer look at the previous method.

In the previous method, records that satisfy the query conjunction are read off the tracks and stored in a sequentially accessed buffer. During the time that a DIP is comparing a record's keywords with the predicates of a query conjunction, the part of the record which has moved past the read head is stored in the buffer in case of a successful comparison. Now, if the comparison fails in the middle of a record, then the part of the record already stored in the buffer must be discarded. This is done by sequenitally moving the buffer memory to restore it to its position at the time the record first appeared at the read head. The buffer is essentially unavailable for input during the recovery process. The nonavailability of the buffer can force the DIP to postpone the processing of a record by one revolution. Thus, in many cases, more than one revolution of the disk device will be necessary to execute a retrieve request. This problem does not exist in the new method for doing retrieves.

6.2 Handling Updates

If the request happens to be an update request, DIP will begin to search the records in the buffer, from end to end, to find those that satisfy the query conjunction in the given request. DIP will set the update bits corresponding to all records that qualify for update. Simultaneously, CIP also begins to search the records in the buffer. Whenever CIP encounters a record whose update bit has been set, it updates the record and then resets the update bit corresponding to the record. Once again, CIP and DIP operate in a pipeline fashion -- that is, even as DIP is reading a record and checking it for satisfaction with the query conjunction, CIP is updating a record that has been previously read from the buffer by DIP. We estimate the time taken to perform an update as approximately two readthroughs. This compares very favorably with the time of three or more revolutions which are necessary in the old method of doing updates (where the records were read off the tracks and not from the buffer). In the old method, one or more revolutions are needed to retrieve all relevant records into the buffer, another revolution is needed to perform the update, and a third revolution is needed to write back the updated records. The algorithms executed by CIP and DIP have been presented in detail in Section 5.

6.3 Handling Insertions

As we recall, from Section 3, TIPs do not do any processing on encountering an insert record request. MMC places the record into the buffer of the TIP corresponding to the track for insertion, even as the TIPs continue their processing of other requests. One revolution is needed, however, to write these records onto the tracks.

The old method of doing insertions, as proposed in Section 3, required the use of two buffers. The first buffer (Buffer B) was used to store records for insertion. The second buffer (Buffer A) was used during the execution of retrieve and update queries in order to store records to be retrieved and sent to the post processor (PP) or to store records to be updated. The old method of doing insertions had one other disadvantage which we shall explain by means of an example. Consider that the following two requests are issued, one after the other, to the same cylinder. The first request is an 'insert-record' request with record R1 as argument. The next request is a 'retrieve' request which is accompanied by query conjunction Qi. Let us assume that record R1 satisfies query conjunction Qi. Therefore, the retrieve request should cause the retrieval of record R1 along with the other records on the track that satisfy Qi. However, since the retrieve request only looks at records in the track and not at records in Buffer B, R1 will not be retrieved.

In the new method of doing insertions, only one buffer is needed with each TIP. This buffer is the size of a track and contains all the records in the track. As records come in for insertion, they are placed in the buffer after the last record in it. A retrieve request will cause DIP to search the buffer for records that qualify for retrieval. Since the buffer also contains all the records that were inserted by 'insert-record' requests that preceded this retrieve request, any record inserted by a prior insert request which qualifies for retrieval will be retrieved. Thus, the disadvantage of the old method has been eliminated.

6.4 Handling Deletions

Each deletion request takes one readthrough to execute. DIP will search the records in the buffer, from end to end, to find those that satisfy the query conjunction in the given request. It will also set the delete bits of all records that qualify for deletion. CIP does no work in performing the delete.

Additionally, we need to do compaction at the end of every set of orders. Earlier, in Section 4, we had shown how to do this in three revolutions. In the first revolution, DIP reads all the untagged records into one buffer and all the tagged records into another buffer. The same revolution is utilized by the CIP to process all the tagged records and form a tagged deletion table (TDT). In the second revolution, DIP writes back all the untagged records while CIP processes the untagged records and forms an untagged deletion table (UDT). Finally, in the third revolution, MMC processes both TDT and UDT to form the structure memory deletion table (SMDT) and it also forms the structure memory insertion table (SMIT).

We shall now propose a scheme for compaction that takes only one readthrough and one revolution and which needs only a single buffer. Recall that all the tagged and untagged records are already in the track-size buffer. In the first readthrough, DIP processes all the tagged records and forms a TDT while CIP processes the untagged records to form a UDT. In the second revolution, all the untagged records in the buffer (this includes the records for insertion) are written onto the track by DIP, while MMC creates SMDT and SMIT. This will, of course, further improve the throughput of the data loop of DBC.

6.5 Execution Times of Various Orders

To summarize, TIPs take one readthrough to execute a retrieve request, one readthrough to execute a delete request, two readthroughs to execute an update request, and no time at all to execute an insert request. Also, one revolution is needed at the beginning of a set of orders to read the records into the track-size buffer, and a revolution and a readthrough are needed at the end of a set of orders to compact and write back the records from the buffer onto the track. Let us estimate the time taken to execute a set of orders given the following parameters.

k=number of orders in the set
p1=percentage of retreive requests
p2=percentage of update requests
p3=percentage of delete requests

p4=percentage of insert requests
r=time for one revolution of the disk device
rt=time for one readthrough of the sequential buffer
T(k)=time to execute a set of orders of length k

Therefore,

$$T(k)=k*[p1*rt + p2*2*rt + p3*rt] + r + r + rt$$
$$=k*rt*[p1 + 2*p2 + p3] + 2*r + rt$$

The value of k (the minimum number of orders that must exist on a cylinder before a seek is initiated on it) is a design choice. We shall explain how the designer (often called a database administrator or DBA) might arrive at a suitable value for k. First of all, he must have an idea of the kind of environment that DBC will operate in. For example, he must know if the environment is update-intensive or not. In terms of the parameters of the previous paragraph, the designer must know, or be able to guess, the values of p1, p2 and p3. Also, he must know the shift-rate of the sequential track-size buffer and the readout rate of the disks of MM. That is, he must know r and rt. Therefore, the only unknown in the equation

$$T(k)=k*rt*[p1 + 2*p2 + p3] + 2*r + rt$$

is k. We recall, from Section 3, that for stability of the data loop the inequality $T(k)/(k*n) < 1$ must be true. By putting $T(k)/(k*n) = 1$, we will arrive at the minimum inter-arrival rate (n) that can be sustained by the data loop in terms of k. That is, the designer has an equation between n and k. Depending upon the value of n that he wishes to have (that is, depending on the throughput rate that he wishes DBC to have), the designer can arrive at a value for k.

6.6 The Components of a TIP

Each TIP consists of the following sub-components.

(1) The disk interface processor (DIP).
(2) The controller interface processor (CIP).
(3) The sequentially accessed query buffer.
(4) The sequentially accessed track-size buffer for the records
(5) A RAM for the delete bit map.
(6) A RAM for the update bit map.
(7) A RAM for the communication area between CIP and DIP.
(8) A RAM to store TDT.
(9) A RAM to store UDT.
(10) A RAM to store SMIT.

The actions performed by CIP and DIP have been explained throughout this report. Primarily, DIP is responsible for reading information in the track-size buffer and for receiving/transmitting data to the tracks of the disk. The main responsibility of CIP is for communicating with the mass memory controller (MMC) over IOBUS. Such communication involves the acceptance of orders and database objects from MMC and transfer of data (retrieved by DIP) and of various tables (created by DIP and CIP) to MMC.

The query memory is a sequential access memory with a capacity to store the largest single query conjunction that may be encountered by MM (about 1 Kbytes). The track-size buffer is also a sequential access memory. This memory is divided into a number of individually accessible segments. Each segment may be read out of or written into in a sequential manner. The motivation for dividing the record buffer into segments is as follows: while DIP is extracting information from the track and placing it in one of the segments, CIP can be transmitting previously extracted information present in one of the other segments to MMC.

Each TIP utilizes two bit maps, the delete bit map and the update bit map, which are small random access memories. Each record on the track has a unique delete bit and a unique update bit. Assuming that the size of a track is 30 Kbytes and that the average size of a record is 200 bytes, we see that the number of records per track is about 150. Thus, each bit map has 150 bits in it. Before processing of a cylinder is to begin, all the bits in the two bit maps are reset. The delete bit is used in the execution of a delete request. When a record is to be deleted, the corresponding bit in the delete bit map is turned on. Subsequent retrieve and update commands will ignore those records that have their corresponding delete bits set. The update bit is used during the execution of update and retrieve requests. When a record is to be udated or retrieved, the corresponding update bit is turned on by DIP to indicate to CIP that the record must be retrieved or updated.

The communication area buffer is also a small random access memory. It contains one bit for each segment in the track-size buffer. If the bit corresponding to a segment is set, then it is an indication to CIP that DIP has finished processing that particular segment. This enables CIP and DIP to operate in a pipeline fashion.

Finally, each TIP has random access memories to store TDT, UDT, and the SMIT. A view of the organization of a TIP is shown in Figure 39.

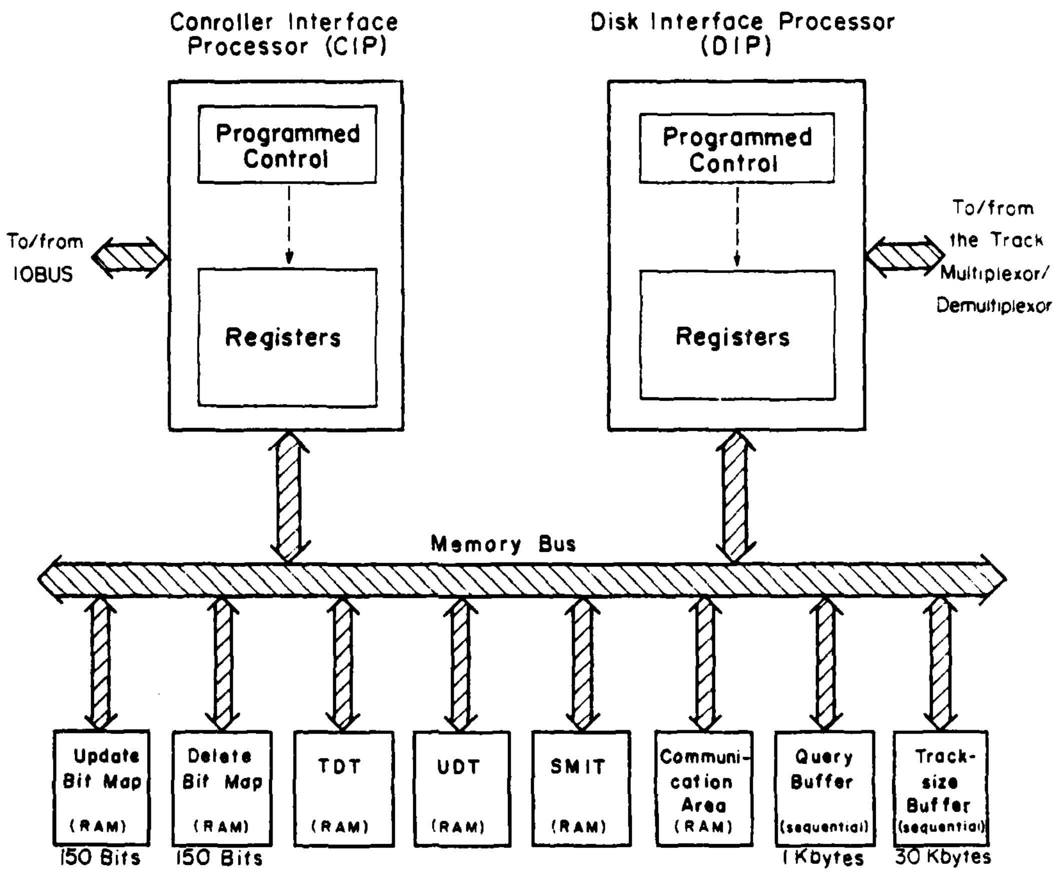


Figure 39. Organization of a Track Information Processor (TIP)

7. CONCLUDING REMARKS

Earlier reports have indicated how DBC handles search and retrieve requests. In this report, we have shown how the process of update is carried out in DBC. Since DBC might often have to be used in an update-intensive environment (that is, an environment where many update, delete and insert commands and only a few retrive commnands are issued), we have indicated, throughout this report, the kind of architectural enhancements which will provide good performance in an update-intensive environment.

Perhaps the most important enhancement that affects the performance of all four types of requests in DBC (retrieve, delete, insert and update) is the incorporation of a track-size buffer with each TIP. A set of orders on a particular cylinder is handled as follows in DBC. In the first revolution of the disk device, the TIPs will read all the records into these track-size buffers. Subsequent retrieve, insert, delete and update commands are executed by reading the records from this buffer rather than off the tracks.

The most important enhancement for insertion requests, is the addition of the insert-in-parallel capacity. That is, records do not have to be inserted into MM of DBC one record at a time. Rather, all the TIPs can be inserting records at the same time.

With respect to deletion, we have shown how to speed up the process of compaction dramatically. In fact, this process which originally took 487 revolutions of the disk device, now takes only one revolution of the disk device and one readthrough of the buffer. This performance improvement is achieved by using a track-size buffer with each TIP, thus avoiding the need to transfer records to and from MMC.

With respect to updates, we have shown how an update request can be handled in two readthroughs of the sequential track-size buffer. This is a substantial improvement over the 16 revolutions that will be necessary to do an update without the use of track-size buffers. Also, the concept of clashing has been introduced. That is, all those requests which will be blocked by an earlier update, which has not been completely executed, are clearly identified. The execution of the blocked requests must be stayed until the blocking update is executed completely.

Throughout the report, we have always substantiated our claims of performance improvement by using an analytical model to come up with some quantative figures of the data loop throughput. The analytical model used is a two node network model, where the first node consists of as many servers as there are disk drive controllers, each with exponentially distributed service times and exponential arrival rates, and the second node consists of a single server with an exponentially distributed service time. By using this model, we have also shown how a data base administrator (DBA) can control the throughput achieved in DBC.

REFERENCES

[1] Banerjee, J. and Hsiao, D. K., "Perforance Evaluation of a Database Computer in Supporting Relational Databases," Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, Federal Republic of Germany, September 1978, pp. 319-329; and Banerjee, J and Hsiao, D. K., "The Use of a Database Machine for Supporting Relational Databases," Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, New York, August 1978, pp. 91-98; also available in Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," Technical Report OSU-CISRC-TR-77-7, The Ohio State University, Columbus, Ohio, November 1977.

[2] Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of ACM '78 Conference, December 1978; also available in Banerjee, J., Hsiao, D. K., and Kerr, D. S., "DBC Software Reqeuirements for Supporting Network Databases," Technical Report OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

[3] Banerjee, J., Hsiao, D. K., and Ng, F. K., "Data Network - A Computer Network of General-Purpose Front-End Computers and Special-Purpose Back-End Database Machines," Proceedings of International Symposium on Computer Network Protocols, (Danthine, A., Editor), Liege, Belgium, February 1978, pp. D6-1 to D6-12; also available in Hsiao, D. K., Kerr, D. S., and Ng, F. K., "DBC Software Requirements for Supporting Hierarchical Databases," Technical Report OSU-CISRC-TR-77-1, The Ohio State University, Columbus, Ohio, April 1977.

[4] Banerjee, J. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4, December 1978, pp. 347-384. Also available in Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part I: Concepts and Capabilities," Technical Report OSU-CISRC-TR-76-1, The Ohio State University, Columbus, Ohio, September 1976.

[5] Kannan, K., Hsiao, D. K. and Kerr, D. S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, October 1977, Niagara Falls, New York, pp. 71-79; and Hsiao, D. K., Kannan, K., and Kerr, D. S., "Structure Memory Designs for a Database Computer," Proceedings of ACM 77 Conference, October 1977, Seattle, Washington, pp. 343-350; also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part II: The Design of the Structure Memory and its Related Processors," Technical Report OSU-CISRC-TR-76-2, The Ohio State University, Columbus, Ohio, October 1976.

[6] Kannan, K., "The Deisgn of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-50; also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part III: The Design of the Mass Memory and its Related Processors," Technical Report OSU-CISRC-TR-76-3, The Ohio State University, Columbus, Ohio, December 1976.

[7] Banerjee, J., Hsiao, D. K., and Menon, J. M., "The Clustering and Security Mechanisms of a Database Computer (DBC), "Technical Report OSU-CISRC-TR-79-2, The Ohio State University, Columbus, Ohio, April 1979.

[8] Banerjee, J. and Hsiao, D. K., "Parallel Bitonic Record Sort - An Effective Algorithm for the Realization of a Post Processor," Technical Report OSU-CISRC-TR-79-1, The Ohio State University, Columbus, Ohio April 1979.

[9] Hsiao, D. K. and Menon, J. M., "The Post Processing Functions of a Database Computer," Technical Report OSU-CISRC-TR-79-6, The Ohio State University, Columbus, Ohio, July 1979.

[10] Bremer, J. W., "Hardware Technology in the Year 2001," Computer, Vol. 9, No.12, December 1976, pp.31-36.

[11] Copeland, G. P., Lipovski, G. J. and Su, S. Y. W., "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proceedings of the First Annual Symposium on Computer Architecture, December 1973, pp. 121-128.

[12] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP - An Associative Processor for Database Management," Proceedings of AFIPS, National Computer Conference, Vol. 44, 1975, pp. 379-387.

[13] Hoagland, A. S., "Magnetic Recording Storage," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1283-1289.

[14] Ampex Corp., PTD-930x Parallel Transfer Drive, Product Description 3308829-01, October 1978.

[15] Hsiao, D. K., Kannan, K., "Simulation Studies of the Database Computer," Technical Report OSU-CISRC-TR-78-1, The Ohio State University, Columbus, Ohio, February 1978.

[16] Chandy, K. M. and Sauer, C. H., "Approximate Methods for Analyzing Queueing Network Models of Computer Systems," Computing Surveys, Vol. 10, No. 3, September 1978, pp 281-317.

[17] Kleinrock, L., Queueing Systems I, John Wiley, New York, 1975.

[18] Su, S. Y. W. and Emam, A., "CASDAL: CASSM's Data Language," ACM Transactions on Database Systems, Vol. 3, No. 1, March 1978, pp. 57-91.

APPENDIX 1

The algorithm presented below is executed by the IP on receipt of an order from the DBCCP.

ALGORITHM 1: To process an MM order from the DBCCP

INPUTS: Input MM order from the DBCCP in the format shown in Figure 6 and the database object used as argument of the order.

Step 1: Use the database object identifier to search the DODT. If the object is already in the DODT, then increment the usage count and go to Step 4.

Step 2: Allocate space for the database object in the DODT. If no space is available, then reject the order and terminate.

Step 3: Place the (sorted) object in the DODT in the block allocated in Step 2. Set usage count to 1.

Step 4: Check the QHT to determine if there is a queue for the MAU referenced by the argument order. If there is a queue, then check if the MAU is being processed currently. If so, go to Step 7. If there is no queue then also go to Step 7.

Step 5: [Order may be added to the queue.] Check if there is a free entry in the OT. If not, go to Step 6. Else, enter the order into the OT and link it to the queue for the MAU. Terminate.

Step 6: [No space in OT.] Reject the order; reduce the usage count of the database object in the DODT. If the usage count is zero, then release space occupied by the object. Terminate.

Step 7: [New queue to be created.] Scan QHT for a vacant entry. If no vacant entry is found, go to Step 6. Else, call the entry number 'p'.

Step 8: Place the MAU address referred to by the argument order in the appropriate field of QHT[p] (see Figure 5). Clear the status bits of QHT[p]. Go to Step 5.

This algorithm is executed by the MMM. It continuously monitors the QHT with a view to keeping the TIPs and the disk drives busy.

ALGORITHM A: To scan the order queues continuously.

Input Argument: QHT

Step 1: [Initialize] p=0.

Step 2: p=p+1. If p > N, then p=1. [N is the number of entries in the QHT.]

Step 3: If QHT[p, 3]='0', then go to Step 2; else, go to Step 4.

Step 4: If QHT[p,0]='0', then go to Step 5; else, go to Step 7.

Step 5: [Initiate access to MAU.] MAUADDR=QHT[p, 8-232]. Decode MAUADDR into disk drive controller number d, drive number k and cylinder number c.

Step 6: Interrogate disk drive controller number d, to determine if the drive k is free. If it is free, then issue a cylinder seek on drive k for cylinder c and set QHT[p, 0] to '1'. Go to Step 2.

Step 7: If QHT[p, 1]='0', then go to Step 8; else, go to Step 10.

Step 8: [Check if seek is complete.] MAUADDR = QHT[p, 8-23.] Decode MAUADDR into drive controller number d, drive number k and cylinder number c.

Step 9: Inerrogate drive controller d to determine if seek on drive k has been completed. If so, then set QHT[p, 1] = '1' and go to Step 10; else, go to Step 2.

Step 10: [Initiate processing if necessary.] If QHT[p, 2] = '0', then go to Step 11; else, go to Step 2.

Step 11: Interrogate if IDLE flag is on to determine if the TIPs are idle. If so, then go to Step 12; else, go to Step 2.

Step 12: [TIPs are idle.] Invoke Algorithm B with the following arguments: number of MAU orders given by QHT[p, 4-7], address of the first order stored in the OT for the MAU and given by QHT[p, 24-39]. Go to Step 2.

In Steps 4 through 6, we try to initiate cylinder seeks for MAUs which have not been accessed so far. In Steps 7 through 9, we check on seeks already issued during a previous scan. In Steps 10 through 12, we try to initiate the TIPs by invoking Algorithm B.

Algorithm B is also executed by the MMM. It is responsible for the detailed dialogues with the TIPs after Algorithm A has found a MAU that has been accessed and is ready to be processed.

ALGORITHM B: To initiate the execution of orders by the TIPs and to accept data retrieved by the TIPs.

Input Arguments: 1. The number N of orders pending execution.
2. The address of the first order in the order table (OT).

Step 1: [Initialize] p=1. FLAG=0.

Step 2: Pick up the p-th order from the OT. If the order code indicates an insert-record order, go to Step 6. If the other code indicates a delete-record order, then go to Step 5. If the order code indicates an update order, then go to Step 7. If the order code indicates a compaction order, then go to Step 15.

Step 3: [Retrieve] Broadcast the order to all the TIPs and go to Step 14.

Step 4: Wait until TIP interrupt occurs. When the interrupt occurs, go to Step 8.

Step 5: [Delete] Broadcast the order to all the TIPs. Turn on DELETE flag. Go to Step 4.

Step 6: [Insert] Broadcase the order "find available space in the track" to all TIPs over the IOBUS. Turn on INSERTION flag. Go to Step 4.

Step 7: [Update] Broadcast the order to all the TIPs. Turn on the UPDATE flag. Go to Step 4.

INTERRUPT ENTRY

Step 8: If the UPDATE flag is on, go to Step 9. If the INSERTION flag is on, then go to Step 10. If the DELETE flag is on, then go to Step 11; else, go to Step 14.

Step 9: [This part of the algorithm is described in Section 5.]

Step 10: Turn off the INSERTION flag. Read, from each TIP, the amont of space available. Choose the track with the largest amount of space available and issue the insert-record request to the TIP corresponding to that track. Also, place the record to be inserted into the buffer of the TIP corresponding to the track chosen for insertion. Go to Step 4.

Step 11: [Check if there was any deletion.] Turn off the DELETION flag. If the TIPs indicate that some records were tagged for deletion, then go to Step 12, else go to Step 13.

Step 12: Store the MAU address in the mass memory deletion table (MMDT).

Step 13: Delete the order from the OT. p=p+1. If p>N, then request the TIPs to write back all deletion tags, set IDLE flag on and halt: else go to Step 2.

Step 14: [Receive retrieved records] If the TIPs have records to be output, then receive them and send them to the SFP. Go to Step 13.

Step 15: [Compaction] Request the TIPs to read all tagged records.

Step 16: As the TIPs transmit tagged records over the IOBUS, create a structure memory deletion table (SMDT), a view of which is shown in Figure 23. Each entry in this table has two fields. The first field contains a directory keyword, and the second field contains a set of pairs, where each pair is of the form (cylinder number, security atom number). Since this table is temporary and may be deleted at the end of the compaction mode, it may be created in the database object descriptor table (DODT). The SMDT is formed as follows. For each tagged record that is received by the MMC, do the following. For each directory keyword DKi in the record, look to see if there is an entry in the SMDT with DKi in Field 1. If no such entry exists, then create a new entry with DKi in Field 1, and the pair (M,S) in Field 2. [M is the cylinder being compacted and S is the security atom number of the record being examined (the seucrity atom to which a record belongs is part of the information present in a record)]. If such an entry exists, then search the corresponding set of pairs in Field 2 of this entry to see if a pair of the form (M,S) exists. If such a pair exists, then do nothing. Else, add the pair (M,S) to the set of pairs in Field 2 of the corresponding entry. Now discard the record. [In the SMDT, we are putting those SM entries which point to records marked for deletion. Since the records have been deleted, these entries (which point to the deleted records) must also be deleted from the SM.]

Step 17: Request the TIPs to read the untagged records. [Since the memory available to the mass memory monitor (MMM) is smaller than the MAU capacity, the MMM will divide the TIPs into secitons which are processed sequentially. Thus, if say 80K bytes are available to the MMM and the MAU capacity is 320K bytes, then the TIPs are divided into four sections. TIPs in the same section are requested to read their tracks concurrently during the compaction process. Steps 18 through 20 are repeated for each section].

Step 18: As the records from the TIPs come in, store them in the record storage.

Step 19: For each record in the record storage, do the following. For each directory keyword DKi in the record, look to see if there is an entry in the SMDT with DKi in Field 1. If no such entry exists, then do nothing. If such an entry exists, then search the corresponding set of pairs in Field 2 of this entry to see if a pair of the form (M,S) exists. M is the cylinder being compacted, and S is the security atom number of the record being examined. If such a pair exists, delete the pair from Field 2 of the SMDT. Else, do nothing. [In this step, we look at those entries in the SM which point to the untagged (undeleted) records. We wish to delete, from the SM, all entries that point to deleted records. However, some entries in the SM may point to both deleted and undeleted records. We wish to retain these entries in the SM, since otherwise, pointers to undeleted records will be lost. Hence, any entry in the SMDT which points to an undeleted record is removed from it.]

Step 20: Write the records in the record storage back into the tracks via the TIPs.

Step 21: Pass the SMDT table to the database command and control processor (DBCCP). (DBCCP then accesses the structure memory (SM) and deletes the relevant index terms from it.) Terminate.

DATE
ILMED

8